Evaluating the Seeding Genetic Algorithm

Honours dissertation of Benjamin Meadows

University of Auckland
Department of Computer Science

UPI: `bmea011`

June 2012

Supervisor: Dr Patricia Riddle

Contents

[1.] Introduction

This work is largely motivated by the PhD thesis of Cameron Skinner [Skinner, 2009], which features a rigorous mathematical and empirical approach to understanding the underlying mechanism behind the functioning of the genetic algorithm. The results are a new understanding of the algorithm in terms of the notions of *discovery*, *selection* and *combination*.

Skinner uses these notions to create a modification to the genetic algorithm: the "seeding" genetic algorithm.[1] We recognise this innovation as an important contribution to the field of evolutionary algorithms, and our focus in this dissertation will be to test its successes, failures, and the scope of its applicability.

[1.1] Background on Genetic Algorithms

[1.1.1] Basic Concepts

The genetic algorithm (GA) is a heuristic search technique in the domain of *evolutionary algorithms*: algorithms which solve problems by mimicking biological evolution.

In broad terms, a genetic algorithm functions by evolving a population of candidate solutions represented in some encoding. Candidates are assessed by a fitness function which measures their quality as a solution, and are selected for reproduction based on this assessment. The offspring of reproducing candidates form the next generation of the population.[2]

As [Koza, 1995] says, the goal of the genetic algorithm is "to find an artificial chromosome which, when decoded and mapped back into the search space of the problem, corresponds to a globally optimum (or near-optimum) point in the original search space of the problem." The basic structure that achieves this is given as pseudocode in figure 1.a. The main operators involved are *selection*, *crossover*, and *mutation*.

Selection is the choice of a candidate solution according to some fitness function specific to the problem, and a selection operator defining the stochastic method by which a candidate with a calculated fitness is chosen. Rank selection orders the candidates by fitness score, then selects candidates with probability proportional to their rank. Tournament selection randomly samples the population for *n* candidates, and chooses the fittest from the sample. Fitness proportional ("roulette wheel") selection chooses candidates with probability proportional to their fitness score relative to that of the population.

Crossover is the creation of new individuals via sexual recombination of other candidates – most commonly two 'parent' solutions. Crossover generates new offspring by selecting genetic material from each parent according to some metric. We will consider uniform crossover, which randomly chooses one

1 Skinner goes on to develop the more abstract "DISCO" algorithm from the seeding algorithm, but we do not discuss it in this work.

2 Typically the population is produced anew as a 'generation' with each iteration of the algorithm, but some variations use steady-state populations.

of the parents to provide each character of genetic information, and *n*-point crossover, which copies from one parent before swapping at *n* points in the string.

Mutation modifies an existing candidate by performing a mutation operation on one character of its code: usually either cycling the character, or randomly setting the character.[3] This typically occurs with some frequency for each character in each candidate; sometimes the entire candidate string must first be selected for mutation according to some other probability.

```
Genetic Algorithm:
[
      initialise population with random candidates
      while termination criteria is not met
      [
            evaluate fitness of population
            select parent candidates from population based on fitness
            with some probability x perform crossover on parents
            with some probability y perform mutation on offspring
            create new population
      ]
]
```

Figure 1.a: Structure of the (Canonical) Genetic Algorithm.

The GA developed from the field of 'artificial evolution'. Its first appearance in the literature in a recognisable form is usually held to be in [Holland, 1975]. The algorithm has been refined considerably in the years since. [Srinivas & Patnaik, 1994] lists many variations, including: extensions of the biological metaphor, methods to reduce sampling errors, adaptive variation of control parameters and dynamic variable encoding, fitness scaling, steady-state GAs, distributed and parallel GAs, and 'messy GAs'.

Increasingly popular are *hybrid* GA systems, utilising genetic algorithms alongside some other problem solver, optimiser or search engine. These systems are often specific to one particular purpose or application. Therein lies the problem: there appears to be no “general” genetic algorithm which can be applied to a broad problem set with a predictable or reliable probability of success. We suspect that the fundamental or *canonical* GA is seldom used in practise today, and if it is, it is only on a few known problems with 'reasonable' parameters established by trial and error.

Perhaps the core problem for the GA, aside from lack of foreknowledge about the best parametrisation of the algorithm, is maintaining population diversity. See, for example, [Luke, Balan & Panait, 2003]. Numerous techniques have been demonstrated to combat early convergence, including crowding [De Jong, 1975], clearing and speciation [Pétrowski, 1996], niching and fitness sharing [Darwen & Yao, 1996], the island model [Whitley, Rana & Heckendorn, 1998], and spatially distributed populations [Sarma & De Jong, 1997]. However, few of them provide a strong guarantee that they will indefinitely prevent premature convergence.

3 When a binary string is used, this is *bit-flip mutation* in the former case or *random mutation* in the latter case.

Skinner's seeding algorithm makes such a guarantee.[4] We consider the efforts of [Skinner, 2009] to also be groundbreaking in systematically confronting questions such as "how does a GA *actually* work, rather than how *should* it work?" and "given the observed failings of the GA, how can we make remodel the way it works to improve it?" We first give a theoretical basis for Skinner's work.

[1.1.2] Theory: Schemata, Building Blocks and GA Difficulty

The operation of genetic algorithms is generally understood by the schema theorem [Holland, 1975]. A *schema* is a template of a string with a certain length. Some of its indices have defined characters and others do not. A string is said to *match* a schema if the defined characters of that schema are the same as those at the same indices in the string.

The "building block hypothesis" of [Goldberg, 1989] states that

(1) *Building blocks* are schemata with *low order* (few defined characters), *low defining length* (defined characters are clustered spatially), and *above average fitness*; and

(2) Genetic algorithms function by heuristically exploiting the existence of building blocks to iteratively build strings of increasing fitness.

With this understanding of the method by which the genetic algorithm solves problems, what constitutes a GA-difficult problem? Two relevant notions are *epistasis*, the interdependence between subsets of characters in the solution encoding, and the *fitness landscape* of a problem, the navigability of the search space topology.

Neither of these are easily quantified, but there are more accessible difficulty metrics. Trivially, a problem space with more building blocks should be more difficult for the GA than one with fewer blocks.[5]

Another measurable metric is *deceptiveness*. Deceptive problems are designed to lead any hill-climbing-style search away from global optima towards local optima. However, [Srinivas & Patnaik, 1994] sum up current understanding by saying that while a deceptive function typically causes difficulty for GAs, deception is neither a sufficient nor a necessary condition for a problem to be GA-hard.

[1.2] Introduction to Seeding

We refer to the work of [Skinner, 2009], who (successfully, in our view) conceptually divides the functionality of the genetic algorithm into *selection*, *discovery* and *combination*. It is on discovery which we wish to focus, and particularly on the operator Skinner introduces: *seeding*.

4 A sufficiently large seed pool size will make homogeneity in the seed pool statistically implausible.

5 Assuming the blocks are of the same length. Two inordinately long building blocks may be harder for the GA to discover than a larger number of more reasonable building blocks, even if the latter problem has longer bit-strings overall as a result.

[1.2.1] Theory Base for the Seeding Algorithm

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 [Skinner, 2009, p 9]. 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.[6] In the cases tested, mutation destroys more than 99% of the blocks it discovers [Skinner, 2009, p 65].

The conclusion is that the problem is not *lack of diversity*, but that the genetic operators in play are simply inappropriate for the way the algorithm is meant to function [Skinner, 2009, p 21]. Continued building block supply, however, is vital – especially since the genetic algorithm is almost guaranteed to rapidly complete when the requisite blocks are present *somewhere* in the population in each generation [Skinner, 2009, pp 49-51]. In practise, existing blocks become extinct in the population with disturbing frequency. *Random search* outperforms the common genetic operators in many circumstances [Skinner, 2009, p 68].

Recognising there is a major trade-off between discovery and combination, Skinner proposes the Seeding Genetic Algorithm (SGA), which uses random search at a single initial stage to perform building block discovery.

The SGA adds a *secondary population* of candidate solutions, called the *seed pool*. This pool is generated by an additional step at the start of the algorithm: a large *presample* of random individuals is generated, and those with fitness higher than the average fitness of the search space are used to populate the seed pool. In practise, this means calculating all the fitness scores of every individual in the presample, then taking the *n* highest-scoring candidates and injecting them into the seed pool.

When the SGA is run, candidates from the seed pool have a chance to be brought into the main population via *seeding*. This is carried out during the crossover step. Each parent is replaced by a random individual drawn from the seed pool with some probability *p*.

There are two benefits of this: the seed pool can introduce or re-introduce blocks which are extinct in the main population, and the parts of seed pool candidates' genomes which are not part of any block schemata provide an increase in the diversity of the main population, preventing early convergence.

Skinner [2009, pp 93-94] shows that the SGA maintains a healthy supply of building blocks and non-fitness-related allele diversity in the main population. The SGA is developed into the DISCO algorithm, discarding the biological metaphor entirely. [Skinner, 2009, p 8] The new design takes into consideration what analysis shows is *actually* happening, not an ideal of what the GA *should* be doing if it is mimicking biological evolution.[7]

---

6 Suggesting the possibility that local search over the candidate string might be superior to the mutation operator.

7 And as any biologist would be quick to point out, as an analogy to true evolution, the genetic algorithm only bears the most tenuous resemblance to begin with.

[1.2.2] How we shall Extend this Work

[Skinner, 2009, p 20] 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* do well in those situations. Demonstrating with statistical rigour the degree to which he has succeeded will comprise a significant part of this dissertation.

We also aim to establish more precisely *where* seeding succeeds or fails. This will involve testing the SGA on more problems, and extrapolating from its ability to solve various parametrisations of these problems. We hope this will lead to a set of rules for the use of the seeding algorithm.

The SGA adds several new parameters to the already quite complex regular genetic algorithm. Our testing will therefore have to be quite multivariate. We aim to establish the nature of the optimal parameters for the SGA, in as generalisable way as we can. If the SGA works well on a problem with a sufficiently broad range of different parameters, and if it is able to perform with roughly the same default parameters on a variety of problems without the need for iterative fine-tuning, then we may consider that Skinner's aim of a more flexible GA operator has succeeded.

[1.3] Outline of this Dissertation

We define the six problems that comprise our problem set in section 2. We determine the repeatability of [Skinner, 2009] and establish our own baselines with the CGA in section 3, then perform additional tests considering the SGA with mutation. We make a preliminary investigation of how seeding *can* and *should* be performed, in the general sense, in section 4.

We consider our first parameter, the seeding probability, in section 5. We attempt to discover optimal values for it, problem by problem, by varying it against the presample size.

We experiment on our second parameter, the seed pool size, in section 6. We attempt to determine optimality by varying it against the presample size. From the results in sections 5 and 6 we determine optimal values for our third parameter, the presample size. We draw our conclusions in section 7.

We discovered some incongruous results with one particular problem/parametrisation combination, and examine those results more closely in Appendix B. The *full* (and very extensive) results for optimal parameter determination are given in Appendices C and D. Reading all the appendices is *not* a requirement for understanding this body of work. They are there to provide additional detail on methodology and results.

[1.4] Conventions and Notations

Our terminology largely follows [Skinner, 2009, pp 12-13], from which we derive the following definitions:

Bit *Bit* refers to any variable that is being optimised by the genetic algorithm. We assume that the genetic algorithm is operating on fixed length binary strings; a bit refers to any element of such a string.

Building block A *building block* is a set of dependent bits that contribute to the fitness calculation of an individual, similar to a schemata. A *complete* block is one with the maximum possible fitness.

Individual An *individual* is a string of bits (sometimes called a chromosome). We may sometimes use *bit-string* or *candidate solution* in the same sense.[8]

Mutation *Mutation* is a unary operator that changes the bits of a single individual in some way; generally by choosing a random value for a bit of an individual with some small probability (the mutation *rate*). Throughout this work we exclusively use *random mutation*, as given in section 1.1.1.

Crossover *Crossover* is a binary operator that combines parts of two parent individuals to create (typically two) offspring individuals. The *crossover rate* is the probability that two parents are actually crossed during a reproduction event. The types of crossover we use are given in section 1.1.1.

Combination Building blocks are *combined* in a reproduction event when an offspring contains building blocks ***X*** and ***Y*** that are inherited from different parents, neither of which contains both ***X*** and ***Y***.

We also use the following terminology, which is in keeping with [Skinner, 2009] or other modern literature in the field.

Main population The *main* population of the genetic algorithm is the dynamic primary population which is operated upon each generation.

Seed pool The *seed pool* or seed population is a static pool of greater-than-average fitness individuals, used for seeding.

Destruction A building block ***X*** is *destroyed* (or disrupted) in a crossover or mutation event when the resulting individual does not contain that block.

Completion An algorithm that *completes* finds the solution to a problem within a certain pre-specified number of generations.

Seeding probability The *seeding probability* or *seed probability* determines the likelihood that a parent in a crossover event is replaced with an individual randomly drawn from the seed pool.

---

8 *Bit-string* typically refers to the actual coded string of the abstract 'individual', and *candidate solution* to the decoded solution provided by the individual. However, since there is a bijective function between the individual, the bit-string and the candidate solution, we may use them interchangeably.

Presample The *presample* is the large initial random sampling of the search space from which the seed pool is drawn by selecting the fittest individuals.

We also commonly use the following abbreviations.

GA Genetic algorithm.

CGA Canonical genetic algorithm.

SGA Seeding genetic algorithm (not to be confused with the "Simple Genetic Algorithm" sometimes found in the literature).

OS Our particular instantiation of the One-Sum problem, as defined in section 2.1.

RR Our particular instantiation of the Royal Road problem, as defined in section 2.2.

R2 Our particular instantiation of the Staggered Royal Road problem, as defined in section 2.3.

HRR Our particular instantiation of the Hierarchical Royal Road problem, as defined in section 2.4.

HIFF Our particular instantiation of the Hierarchical If-And-Only-If problem, as defined in section 2.5.

DT Our particular instantiation of the Deceptive Trap problem, as defined in section 2.6.

When we wish to make statistical inferences, we use one of two tests for statistical significance. When considering data in which all runs of the algorithm were observed to complete, we use the standard 2-tailed non-paired Student's *t*-test. Whenever we have "censored" data (data for which one or more runs of the algorithm did not solve the problem within time bounds) we use the bootstrap test described in [Cohen & Kim, 1993]. For the discussion of the reasoning behind the choice of these statistical tools, see Appendix A.

Throughout, when we claim a "statistically significant" difference, we mean that the null hypothesis can be discarded with a P value < 0.05. If we say "highly statistically significant", we mean that the null hypothesis can be discarded with a P value < 0.01.

[2.] Problem Definitions

[2.1] One-Sum (OS)

The GA-easiest problem we could find is the trivial One-Sum, or One-Max, problem [Thierens & Goldberg, 1994], [Giguere & Goldberg, 1998]. One-Sum simply has its solution the string *1...1*. The fitness of an individual is equal to the number of bits set to 1 in its bit-string (the sum of its ones).

This problem is easy to the point of being trivial: there is no deception involved, so selection will favour those individuals with the most 1-bits. Crossover will result in offspring likely to have more 1-bits. The fitness landscape is essentially a cone: every individual with more 1-bits than its neighbour is correspondingly more fit. It is perfect for a hill-climbing-style search of the problem space.

However, note that the problem doesn't have building blocks *per se* (unless we equate each *bit* with a short-defining-length set of bits contributing to the fitness of the individual; in which case there are 64 trivial building blocks). We include it simply as a baseline: if an algorithm fails to solve the One-Sum problem, we know something has gone wrong. Our implementation of this problem, which we call OS, is a 64-bit One-Sum problem.

[2.2] Royal Road (RR)

The Royal Road is a classic case study in the literature. It was designed as a GA-easy problem, but discovered to be unusually difficult [Mitchell *et al*, 1992]. The Royal Road lacks deception, has a single solution, features discrete building blocks on which an appropriate fitness function can be directly based, and can be solved by other optimisation techniques such as hill-climbing. Despite all these positive signs, the genetic algorithm still encounters difficulties with the problem. [Skinner, 2009] uses it as an important baseline, so we focus on this problem in particular.

In the Royal Road problem, the fitness of an individual is equal to the sum of the values of its schemata, where each schema has the same value as its order (the number of defined bits). There are no overlapping bits between schemata, no disagreements between schemata, and typically every bit in an individual is part of (exactly) one schema.

AAAAAAAA*BBBBBBBB*CCCCCCCC*DDDDDDDD*EEEEEEEE*FFFFFFFF*GGGGGGGG*HHHHHHHH*

Figure 2.2.a: Composition of the Royal Road problem implemented as RR. Each position in the genome corresponds to a bit. The letter [A...*H*] indicates the block to which a bit belongs.

Our implementation of this problem, which we shall refer to as RR, is a 64-bit Royal Road problem consisting of eight coherent blocks of eight bits each, adjacent to each other, as shown in figure 2.2.a. This is equivalent to the R_1 problem in [Skinner, 2009, p 16], originally from [Mitchell & Forrest, 1997].

[2.3] Staggered Royal Road Variant (R2)

The Staggered Royal Road is a standard Royal Road variation described in [Mitchell & Forrest, 1997], where each block has the greatest possible distance between its first and last character. The blocks are thus no longer coherent, but have their information maximally staggered across the length of the string.

The version we implement, which we call R2, has eight blocks of eight bits each, arranged as in figure 2.3.a for maximum bit spread. This is the same as Skinner's R_2 [2009, p 16].[9]

Figure 2.3.a: Composition of the Staggered Royal Road problem implemented as R2. Each position corresponds to a bit. The letter [A...H] indicates the block to which a bit belongs. All the bits of the first block (A) are emphasised.

It is obvious that *n*-point crossover should perform poorly on this problem: each crossover event will disrupt seven or eight blocks, not just *at most one* (in the case of 1-point crossover on the RR problem) or *at most two* (in the case of 2-point crossover on the RR problem).

[2.4] Hierarchical Royal Road Variant (HRR)

The Hierarchical variant of the Royal Road problem is a form not considered by Skinner, but described in [Mitchell *et al*, 1992] as the original form which – surprisingly – the genetic algorithm struggled with.

This problem breaks the assumption of the previously-described Royal Road that each bit belongs to at most one schema. Instead, there are blocks of different *tiers* or *orders*. Each completed block has a fitness value equal to its length. Higher-order blocks are therefore worth more to the fitness function.

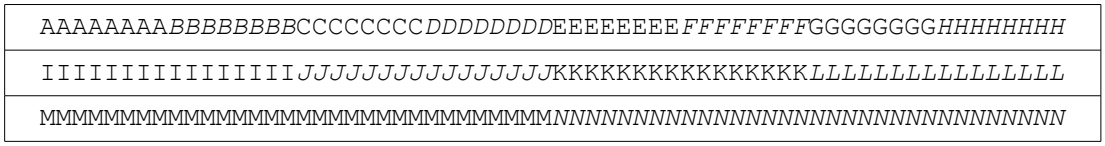

Figure 2.4.a: Composition of the Hierarchical Royal Road problem implemented as HRR. Each position corresponds to a bit. The letter [A...*N*] indicates the block(s) to which a bit belongs. The first row shows the primary blocks. The second row shows the secondary blocks. The third row shows the tertiary blocks.

In our implementation, which we call HRR, there are primary, secondary and tertiary blocks. Primary blocks are the same as the elementary blocks of RR. There are two types of higher-order block. There are four secondary blocks, each comprised of two specific primary blocks. Similarly, there are two tertiary

9 Note that Mitchell *et al* use R_2 to refer to the *Hierarchical* Royal Road, discussed in section 2.4.

blocks, each comprised of two specific secondary blocks. A primary block contributes 8 fitness. A secondary block contributes 16 fitness, in addition to that of its two primary blocks. A tertiary block contributes 32 fitness, in addition to that of its two secondary blocks and four primary blocks. This structure can be seen in figure 2.4.a.

The usual explanation for the GA-difficult nature of this problem is *hitchhiking* (see [Mitchell, Holland & Forrest, 1994]). Hitchhiking refers to the problematic state where higher-order blocks are considered so valuable (in terms of fitness) that individuals containing those blocks are excessively favoured during selection. The non-fit parts of those strings are therefore propagated through the population at the expense of lower-order blocks occupying the same positions.

It will be interesting to see how the SGA performs on this problem, given that it is the original "unusually-difficult" problem for the canonical genetic algorithm.

[2.5] Hierarchical If-And-Only-If (HIFF)

The Hierarchical If-And-Only-If problem is used by [Skinner, 2009, p 16] as a more "interesting" problem with multiple global optima. Described in [Watson & Pollack, 2000], Hierarchical If-And-Only-If is a recursive problem: the bit-string is separated iteratively into halves, and rewarded with a fitness bonus equal to the combined length of the two halves if either:

(a) all the bits of the two halves are set to 1, or
(b) all the bits of the two halves are set to 0.

The same criteria are recursively applied to each new halving, until a string length of 1 is reached.

We implement a Hierarchical If-And-Only-If problem of length 64, which we call HIFF. Figure 2.5.a shows the two possible optimal individuals for the HIFF problem, and the two possible penultimate-fitness individuals.

	0000000000000000000000000000000000000000000000000000000000000000	
	1111111111111111111111111111111111111111111111111111111111111111	
*	1111111111111111111111111111111100000000000000000000000000000000	*
*	0000000000000000000000000000000011111111111111111111111111111111	*

Figure 2.5.a: Composition of the Hierarchical If-And-Only-If problem implemented as HIFF. The first and second rows shows the individuals of maximum fitness. The third and fourth rows, marked with asterisks, show the individuals of penultimate fitness.

Although not deceptive, this is a difficult problem. There are 254 blocks of six different lengths, and there is extremely strong interdependence: the fitness contribution of each block depends on all the others. There are also two possible final solutions, and these are maximally different from each other.

The fitness landscape is therefore essentially *made* of local optima. For each optimum, whether it is global or local, there are two less-fit optima equidistant from it. This is a fractal landscape, and as

[Skinner, 2009, p 17] points out, very difficult for a hill-climbing algorithm to solve. We will see in section 3.1.5 that it is also difficult for the basic GA to solve.

[2.6] Deceptive Trap (DT)

Deceptive Trap, like Hierarchical If-And-Only-If, is designed to make alternative optimisation algorithms such as gradient descent fail; the difference is that Deceptive Trap does so at the level of the building block. In other words, it is *deceptive*. The problem, described in [Goldberg, 1987], is analysed by Skinner [2009], and is the only outright deceptive problem we will investigate.

Each block in the genetic code is a *trap*. Its fitness is equal to its length if all bits are set to 1, or to its number of bits set to 0, minus 1, otherwise. Thus a 'complete' block, with maximal fitness *n* equal to its length, has all bits set to 1, but the block of penultimate fitness has all bits set to 0, and so on.

This is precisely the type of problem we expect the GA to fail on: incremental improvements for a block lead to the algorithm getting stuck in a local maxima such that a *complete inversion* is required to go from the trapped state to the maximum for that block, with the fitness slope to that real maximum leading in the wrong direction every step of the way. Any hill-climbing style search will lead directly away from the global maximum.

AAAAA|BBBBB|CCCCC|DDDDD|EEEEE|FFFFF|GGGGG|HHHHH|IIIII|JJJJJ|KKKKK|LLLLL

Trap: [**11110**]	Fitness = 1
Trap: [**10101**]	Fitness = 2
Trap: [**01001**]	Fitness = 3
Trap: [**00010**]	Fitness = 4
Trap: [**11111**]	Fitness = 5

Figure 2.6.a: Composition of the Deceptive Trap problem implemented as DT. The top row shows the genome for an individual. The five lower rows show examples of traps in increasing order of fitness, demonstrating the deceptive nature of the problem.

Our implementation, which we call DT, is a 60-bit string consisting of 12 traps of length 5 each. We chose this instead of 13 traps of length 5 (making it closer to the 64-bit string we otherwise use) partly because the deception should make it sufficiently difficult, and partly to match the implementation which Skinner used [2009, p 32]. The composition of DT is demonstrated in figure 2.6.a.

[3.] Repeatability of Earlier Work and Determining Baselines

We initially attempted to reproduce the results of [Skinner, 2009]. The RR, R2, HIFF and DT problems are all directly comparable to his work. We additionally tested HRR and OS.

[3.1] Canonical GA Tests

We first ran the CGA on each problem, to provide us with a baseline. The parameters used are summarised in table 3.1.a. These are the same parameters used by [Skinner, 2009]. These values are used for all the experiments described in this dissertation, except where noted.

Runs per experiment	1000
Generations per run	1000
Population	128
Crossover rate	0.9
Mutation rate	1/64
Standard parameters for our experiments	

Table 3.1.a: Standard GA parameters.

[3.1.1] Canonical Test: OS

The 64-bit One-Sum problem was tested first, as the simplest of all the problems in our test set. As expected, the CGA was readily able to solve the problem. The exception, as we can see from table 3.1.1.a, is the case in which fitness proportional selection is used. In this case, the CGA was never able to solve the problem. This is a signal that, in the absence of a special operator to maintain diversity, the population quickly converges to instances of a single individual, from which point mutation alone is insufficient to reach the true solution.

	1-point	2-point	Uniform
2-tournament	1000	1000	1000
5-tournament	1000	1000	1000
32-tournament	1000	1000	1000
Rank selection	1000	1000	1000
Fitness proport.	0	0	0

Table 3.1.1.a: Number of runs out of 1000 in which the CGA solved the OS problem.

From table 3.1.1.b, we see that *n*-tournament selection with high values of *n* appears to be most effective. Uniform crossover was statistically significantly better than 2-point crossover, which was in turn significantly better than 1-point crossover. Additionally, 5-tournament and 32-tournament were statistically significantly better than 2-tournament and rank selection. This holds for all cases other than fitness proportional selection, which we cannot make statistical claims about.

	1-point	2-point	Uniform
2-tournament	46.2 (8.8)	39.5 (7.1)	28.4 (4.5)
5-tournament	20.7 (3.1)	18.1 (2.6)	12.3 (1.8)
32-tournament	15.3 (2.5)	14.2 (2.3)	10.8 (2.2)
Rank selection	44.9 (8.6)	38.5 (7.1)	27.4 (4.3)
Fitness proport.	N/A	N/A	N/A

Table 3.1.1.b: Mean number of generations required for the CGA to solve the OS problem. Standard deviations are given in parentheses (σ).

[3.1.2] Canonical Test: RR

Next, the canonical Royal Road problem. The CGA was able to solve RR in the large majority of runs. From table 3.1.2.a we can see that fitness proportional selection again performed very poorly, and uniform crossover was also troublesome (as expected under the building block hypothesis: uniform crossover is much more likely to disrupt a block than *n*-point crossover).

	1-point	2-point	Uniform
2-tournament	929	951	775
5-tournament	997	1000	1000
32-tournament	999	1000	1000
Rank selection	950	969	860
Fitness proport.	72	74	12

Table 3.1.2.a: Number of runs out of 1000 in which the CGA solved the RR problem.

Similarly to the OS problem, *n*-tournament selection with high values of *n* appears to be the most effective (see table 3.1.2.b).

	1-point	2-point	Uniform
2-tournament	497.1 (206.4)	470.8 (196.7)	626.0 (203.8)
5-tournament	284.3 (145.9)	256.6 (146.4)	246.8 (118.9)
32-tournament	265.1 (137.7)	242.6 (129.9)	193.4 (97.4)
Rank selection	465.1 (209.4)	449.0 (198.6)	577.5 (204.1)
Fitness proport.	580.3 (234.7)	574.3 (239.3)	640.3 (210.3)

Table 3.1.2.b: Mean number of generations required for the CGA to solve the RR problem. Standard deviations are given in parentheses (σ).

Table 3.1.2.c compares these results to those of [Skinner, 2009, p 41]. The figures are, disappointingly, dissimilar. Genetic algorithms are notoriously prone to the 'butterfly effect', whereby disproportionately large changes emerge from slight differences in the initial state or parameters. We note that the overall *pattern* is quite similar between the two sets of experiments.

	Mean generations to complete	
Selection type	**Our work**	**Skinner, 2009**
2-tournament	470.8 (196.7)	214.4 (132.4)
5-tournament	256.6 (146.4)	175.1 (103.5)
32-tournament	242.6 (129.9)	210.7 (111.3)
Rank selection	449.0 (198.6)	208.4 (111.7)
Fitness proportional selection	574.3 (239.3)	474.5 (235.6)

Table 3.1.2.c: Comparison of our baseline results to those of [Skinner, 2009, p. 41], using 2-point crossover. Standard deviations are given in parentheses (σ).

Also consider the charts below. Figure 3.1.2.d shows the mean number of blocks present in 1000 runs of the CGA over the RR problem, using 1-point crossover. Comparing it to table 3.1.2.b, we can see that the worst-performing form of selection, fitness proportional selection, has the fewest blocks present after generation ~100 (the dark red line in the graph). The next-worst performers, 2-tournament and rank selection, make slightly better progress on average (the blue and green lines in the graph) but fall well below 5-tournament and 32-tournament (orange and yellow lines in the graph).

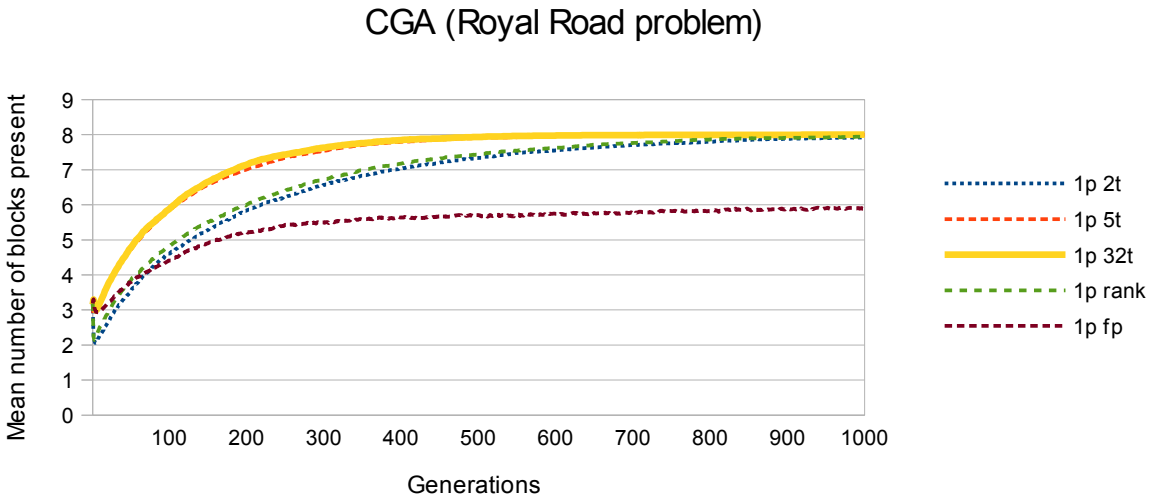

Figure 3.1.2.d: Mean number of blocks present during 1000 runs of the CGA on the RR problem, with 1-point crossover. "2t" refers to 2-tournament selection, "5t" to 5-tournament selection, "32t" to 32-tournament selection, "rank" to rank selection and "fp" to fitness proportional selection.

Figure 3.1.2.e is a magnification of the first 30 generations from the same dataset. We see that 32-tournament and fitness proportional selection, which are the functions most likely to select a candidate which is truly the fittest in the population, immediately make some progress as they rapidly combine the fittest random strings, followed by a slump presumably correlating to convergence. A few generations later, 32-tournament begins to climb out of this slump, and is joined by 5-tournament, while fitness proportional selection improves more slowly. In contrast, the "weakest" selectors[10], rank selection and

10 "Weakest" in that they are the functions least likely to select a high-fitness candidate.

2-tournament, experience an immediate large loss of fitness, presumably caused by failing to select fit strings from the initial population, followed by the beginning of intermediate growth.

This may reflect the idea that once 32-tournament or 5-tournament *do* find an improvement, e.g. a building block in the Royal Road, that improvement is quickly propagated to the rest of the population, whereas with the 'weak' selectors, it is more likely to be lost. Fitness proportional selection is a *very* strong selector, and does poorly: it may be that it is unable to prevent early convergence.

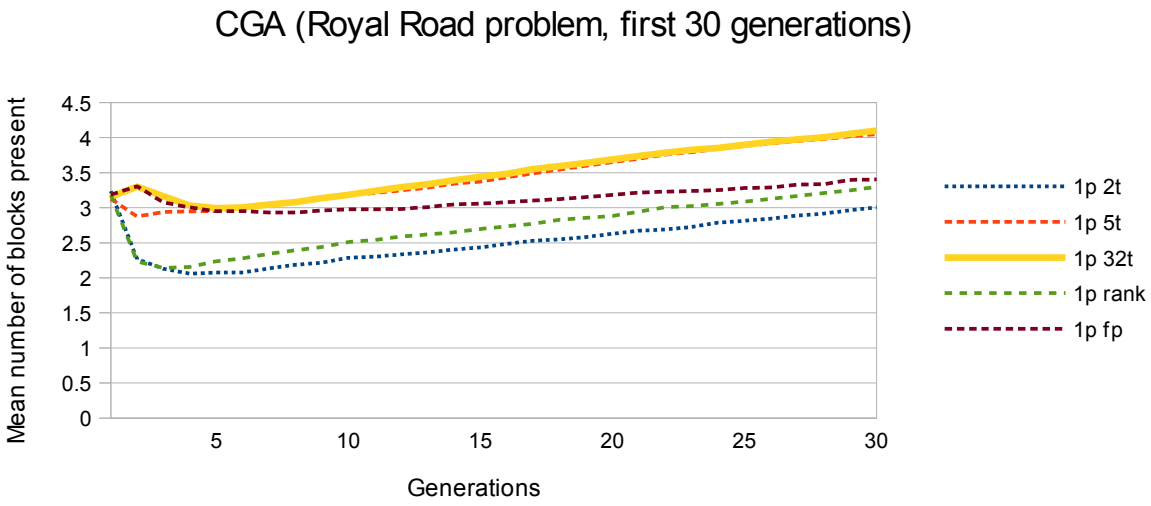

Figure 3.1.2.e: Mean number of blocks present during the first 30 generations of 1000 runs of the CGA on the RR problem, with 1-point crossover. "2t" refers to 2-tournament selection, "5t" to 5-tournament selection, "32t" to 32-tournament selection, "rank" to rank selection and "fp" to fitness proportional selection.

[3.1.3] Canonical Test: R2

For the staggered problem, uniform crossover worked better than either 1-point or 2-point crossover, to a statistically significant level for all but fitness proportional selection. See tables 3.1.3.a and 3.1.3.b. Skinner [2009] only tests 2-point crossover, and has rather unexpected results: R2 with 2-point crossover is *easier* to solve than RR with 2-point crossover, and RR with uniform crossover works better than *either* of the others [Skinner, 2009, p 69].

	1-point	2-point	Uniform
2-tournament	402	482	750
5-tournament	1000	1000	1000
32-tournament	1000	1000	1000
Rank selection	588	673	849
Fitness proport.	3	2	750

Table 3.1.3.a: Number of runs out of 1000 in which the CGA solved the R2 problem.

Our findings partially match these: R2 is either significantly easier than or not significantly different from RR with *n*-point crossover. The distinction between uniform and *n*-point crossover is a little more varied.

Skinner [2009, p 69] postulates that the greater discovery rate of uniform crossover makes up for the disruption it causes by crossover.

	1-point	2-point	Uniform
2-tournament	659.7 (202.1)	660.8 (192.2)	619.6 (203.5)
5-tournament	279.3 (203.5)	264.7 (118.3)	237.1 (115.6)
32-tournament	235.5 (113.5)	219.4 (105.4)	194.8 (92.2)
Rank selection	630.0 (209.5)	603.6 (209.5)	574.8 (206.1)
Fitness proport.	376.3 (161.9)	786.5 (186.0)	738.4 (147.7)

Table 3.1.3.b: Mean number of generations required for the CGA to solve the R2 problem. Standard deviations are given in parentheses (σ).

[3.1.4] Canonical Test: HRR

Even fewer of the runs of the CGA on the HRR problem completed than did for R2 (see tables 3.1.4.a and 3.1.4.b). The algorithm takes statistically significantly longer than RR for all parameters (except fitness proportional selection, which has too few results to compare). Skinner did not consider the problem, so we cannot compare our new baseline, but the difficulty the GA on HRR compared to RR is as predicted by the work of [Forrest & Mitchell, 1993].

	1-point	2-point	Uniform
2-tournament	318	291	80
5-tournament	999	1000	1000
32-tournament	996	998	1000
Rank selection	433	373	123
Fitness proport.	6	4	1

Table 3.1.4.a: Number of runs out of 1000 in which the CGA solved the HRR problem.

	1-point	2-point	Uniform
2-tournament	559.9 (240.0)	542.3 (248.2)	597.1 (240.8)
5-tournament	295.9 (155.2)	268.3 (114.9)	230.5 (115.5)
32-tournament	270.5 (136.4)	252.7 (134.3)	189.2 (92.3)
Rank selection	550.4 (236.5)	539.4 (246.5)	571.4 (238.7)
Fitness proport.	444.8 (335.1)	595.5 (236.1)	296.0 (*)

Table 3.1.4.b: Mean number of generations required for the CGA to solve the HRR problem. Standard deviations are given in parentheses (σ).

[3.1.5] Canonical Test: HIFF

The results of the CGA for the HIFF problem are given in tables 3.1.5.a and 3.1.5.b below.

	1-point	2-point	Uniform
2-tournament	74	73	38
5-tournament	1	1	0
32-tournament	1	2	0
Rank selection	71	92	36
Fitness proport.	0	0	0

Table 3.1.5.a: Number of runs out of 1000 in which the CGA solved the HIFF problem.

As expected, the CGA struggles to solve this 'hard' problem at all. The relatively greater success of the 'weaker' forms of selection (2-tournament and rank selection) is expected: these are the ones most likely to preserve diversity, and once HIFF has converged to an invariant population, the odds of changing the population (getting out of a local optimum) are negligible.

	1-point	2-point	Uniform
2-tournament	601.6 (259.3)	549.9 (258.3)	606.6 (261.1)
5-tournament	58.0 (*)	25.0 (*)	N/A
32-tournament	136.0 (*)	21.5 (17.7)	N/A
Rank selection	574.9 (220.0)	561.12 (249.8)	621.6 (267.1)
Fitness proport.	N/A	N/A	N/A

Table 3.1.5.b: Mean number of generations required for the CGA to solve the HIFF problem. Standard deviations are given in parentheses (σ).

For example, if the algorithm converged at a penultimate-fitness string, half zeroes and half ones, it would require changes in 32 positions to reach the final solution. Each of these changes, taken individually or as a group of < 32, would strictly decrease the fitness of the string. The spontaneous 'switching' mutation of precisely 32 bits is obviously unlikely in the extreme.

Skinner does not succeed with the CGA on the HIFF problem. The analysis [Skinner, 2009, pp 31-32] shows there is almost no combination of building blocks after generation 20 or so, confirming our hypothesis of early convergence.

[3.1.6] Canonical Test: DT

Our experiments with the canonical genetic algorithm on DT resulted in no completing runs at all. [Skinner, 2009, p 36] reports no progress after around 50 generations. His figures show massive destruction of building blocks up to generation 20 or so, after which an equilibrium appears to be reached. Both these observations are mirrored in figures 3.1.6.a and 3.1.6.b below. Note that fitness proportional selection is slowest to lose blocks, but has the greatest overall loss. 5-tournament and 32-tournament are the fastest, but level out with a greater number of blocks retained.

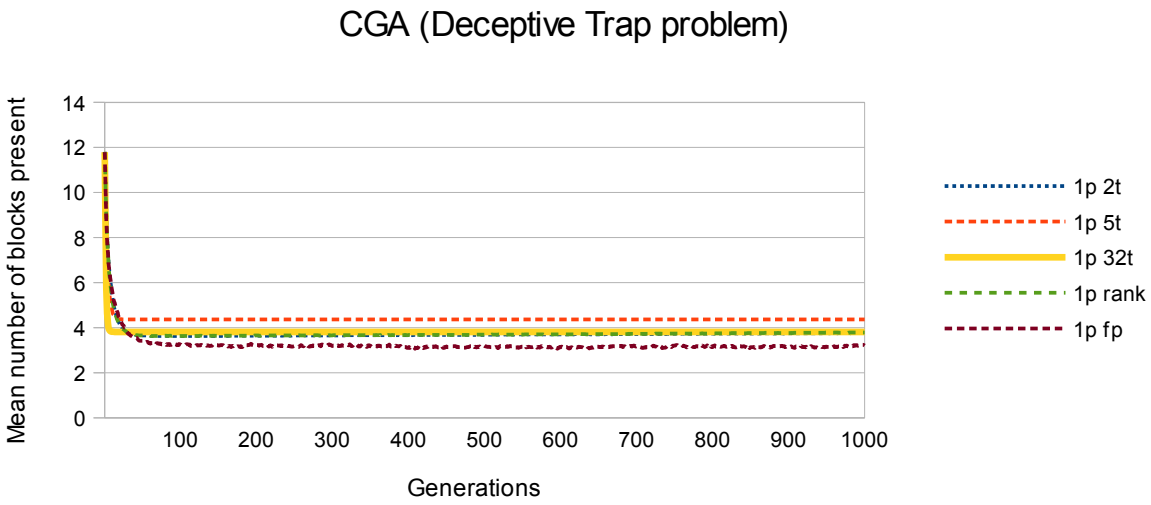

Figure 3.1.6.a: Mean number of blocks present during 1000 runs of the CGA over the DT problem, with 1-point crossover. In the graph, "2t" refers to 2-tournament selection, "5t" to 5-tournament selection, "32t" to 32-tournament selection, "rank" to rank selection and "fp" to fitness proportional selection.

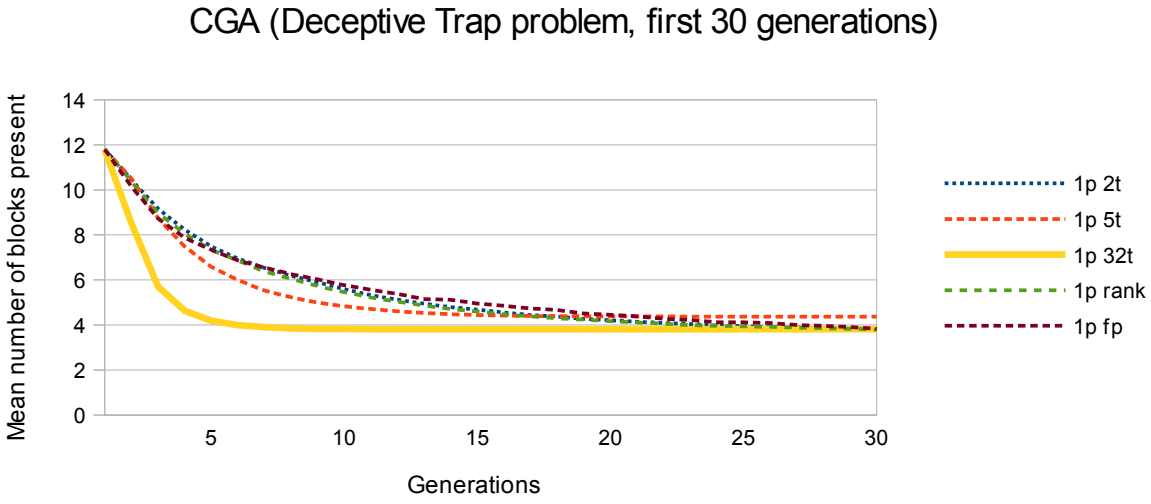

Figure 3.1.6.b: Mean number of blocks present during the first 30 generations of 1000 runs of the CGA over the DT problem, with 1-point crossover. "2t" refers to 2-tournament selection, "5t" to 5-tournament selection, "32t" to 32-tournament selection, "rank" to rank selection and "fp" to fitness proportional selection.

[3.2] Initial Seeding Algorithm Tests on Royal Road (with Mutation)

Skinner's work aims to create a new discovery operator, finding the destructiveness of mutation unsuitable for the task. Our initial tests of the seeding algorithm on the RR problem varied the seeding probability against the mutation rate. We aimed to establish an idea of what a "reasonable" seeding probability is. [Skinner, 2009, p 121] picked the seeding probability 1/3 simply because it meant approximately half of the crossover operations would have at least one parent replaced.

	Mutation (1/64)	**No mutation**
2-tournament:		
0 seed probability (CGA)	491.1 (203.5)	**N/A**
0.1 seed probability	[0 completed]	**203.0 (146.9)**
0.2 seed probability	[0 completed]	708.0 (140.0)
0.3 seed probability	[0 completed]	[0 completed]
5-tournament:		
0 seed probability (CGA)	257.3 (134.5)	**N/A**
0.1 seed probability	**100.6 (67.2)**	**98.6 (91.0)**
0.2 seed probability	***166.9 (132.2)***	***75.5 (71.8)***
0.3 seed probability	***472.7 (289.3)***	***193.1 (159.9)***
32-tournament:		
0 seed probability (CGA)	249.2 (131.1)	**N/A**
0.1 seed probability	***94.67 (66.4)***	***117.1 (103)***
0.2 seed probability	***89.3 (65.1)***	***79.2 (70.8)***
0.3 seed probability	***96.8 (75.7)***	***76.9 (67.9)***
Rank selection:		
0 seed probability (CGA)	451.8 (196.4)	**N/A**
0.1 seed probability	314.3 (349.5)	**177.7 (129.5)**
0.2 seed probability	[0 completed]	379.6 (361.5)
0.3 seed probability	[0 completed]	[0 completed]
Fitness proportional:		
0 seed probability (CGA)	615.9 (226.2)	**N/A**
0.1 seed probability	523.7 (238.2)	**417.2 (267.5)**
0.2 seed probability	[0 completed]	624.4 (204.3)
0.3 seed probability	[0 completed]	[0 completed]

Table 3.2.a: Mean number of generations required for the SGA to solve the RR problem (using 2-point crossover). Values are in *italics* when there is a statistically significant difference between 'with' and 'without' mutation. Values are in <u>bold</u> when there is a statistically significant difference between the performance of the SGA and that of the CGA.

We also wished to apply an experimental framework to the question of whether there is any point to retaining mutation *alongside* seeding; Skinner did not appear to test this, presumably because his theoretical analysis claimed that mutation was an ineffective and inappropriate discovery operator.

We used the standard presample size of 1000 and seed pool size of 50. For these initial tests we used only 100 runs per experiment, and only tested 5-tournament and 2-point crossover. The results varied highly. In testing a wide range of seed probabilities, we found that ~0.15 was optimal when mutation was not being used, 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. It would appear that there is some level of interference between mutation and the seeding operator.

The SGA with seed probabilities in the area between 0.1 and 0.2 appeared to outperform the CGA, and it was at least comparable with seed probabilities as low as 0.05 and as high as 0.3. Outside that range, the SGA effectively failed on this problem. We therefore focus our attention on seed probabilities around 0.1 to 0.2.

We also performed longer initial tests over a larger number of variables, with different types of crossover and selection. The results for 2-point crossover can be seen in table 3.2.a. It appears that the SGA, on the RR problem, is almost always *no worse than* the CGA, and that with a low seed probability it is usually *significantly better*. Mutation typically worsens the performance of the SGA. The parameters chosen do strongly affect the algorithm's performance. Uniform and 1-point crossover give generally similar results to those shown.

[3.3] Comparison to Earlier Work and New Baseline Results

Our early results are reasonably comparable to [Skinner, 2009, p. 93], although we used 1000 runs where he used 100, and his seed probability was 1/3. Skinner found that the SGA outperformed the CGA by 125 generations, or 131 generations if given normal mutation as well. Our most similar test, the "5-tournament" section of table 3.2.a, found that the SGA outperformed the CGA by a region of 160-180 generations at its most optimal. However, this optimal point was not the same as Skinner's.

The differences in our results are somewhat frustrating, but as we were not able to resolve them, we must ascribe them to minor implementation differences causing greater run-on effects (until further work can be done). We have established a baseline of our own to measure relative performance differences.

[3.4] Initial Seeding Algorithm Tests on Other Problems (with Mutation)

We extended our initial tests to see whether a similar ratio of seed probabilities to mutation rate held for the other problems in our problem set. HRR favoured 0.15, and slight mutation, just like RR. The SGA appeared to perform very poorly on R2, especially without mutation. The SGA was able to solve DT, unlike the CGA, and favoured a lower seed probability of 0.1; mutation was largely unhelpful.

The SGA solved HIFF better than the CGA could, although still not *easily*; mutation appeared to be neutral to the success on that problem. The SGA appeared to work best with a higher seed probability of 0.2 on HIFF. The SGA was unable to beat the CGA for the simple OS problem; a hybrid algorithm with seeding and full mutation performed better, but was still worse than the CGA.

Overall, it appears the coupling of mutation to the SGA is not particularly helpful. We will therefore only consider the SGA *sans* mutation for the remainder of this work.

[4.] Investigating the Seeding Process

Our next step was to scrutinise the seeding process itself. As [Skinner, 2009, p 121] notes, a seeding probability of 1/3 will give an approximately 50% chance of replacing *at least* one parent in a crossover event with a seed individual. We have already seen that the performance of the SGA can vary greatly depending on the seed probability.

It makes a certain amount of intuitive sense that seeding could be less or more useful at different points in the progression of the algorithm. For instance, if seeding is providing useful functionality primarily by preventing early convergence, then it might become relatively more useful later in a run. On the other hand, if its main utility lies in being a less-destructive discovery operator, as Skinner believes, it should be useful throughout the entire process.

[4.1] Dynamic Seed Probability Functions

We experiment by varying the seed probability over the course of each run of the seeding algorithm. Because we have witnessed the SGA becoming ineffective with large seed probabilities, we introduce seeding *caps*. We compare three types of function:

- *Static functions*. These are the functions we have used so far. The same flat value of 0.1, 0.2 or 0.3 is used for the seeding probability at every generation.
- *Linear-increasing functions*. We consider functions where the seeding probability is equal to 1% or 0.1% of the generation number, up to a cap of either 10%, 20% or 30%. This cap is reached at generation 10, 20 or 30 in the first case, or 100, 200 or 300 in the second case.
- *Log-decreasing functions*. We consider functions which assign seed probability $0.1 \log(n)$ and $0.075 \log(n)$, where n is the current generation. These are given caps at either 0.1, 0.2, or 0.3. The first log function reaches these caps at generations 10, 100 and 1000 respectively. The second function reaches these caps at generations 22, 465, and (not at all) respectively.

Seeding Function	Seed Probability Formula	Generations to Complete
Static	**0.2 seed probability**	165.9 (144.6)
Fast Linear	**0.01 * generation#**	163.3 (132.9)
Slow Linear	**0.001 * generation#**	*232.1 (142.7)*
0.1 Log	**0.1 * $\log_{10}$(generation#)**	166.5 (135.1)
0.075 Log	**0.075 * $\log_{10}$(generation#)**	*180.0 (139.9)*

Table 4.1.a: Mean number of generations required for the SGA to solve the RR problem (using 5-tournament and 2-point crossover). Seeding probability is capped at a maximum of 0.2. Values are in *italics* when they are statistically significantly different from the static function in the first row.

In table 4.1.a, we present the relevant figures for the cap of 0.2 maximum seed probability[11]. In two cases, the alternative functions are not significantly different from the flat function ($p > 0.6$ for the linear function, $p > 0.9$ for the log function). In two cases they are significantly worse. This was the best performance given by the dynamic functions.

We analyse how the seed probability functions are affecting the population by looking at the mean number of blocks present over the first 100 generations, in each test of 1000 runs.

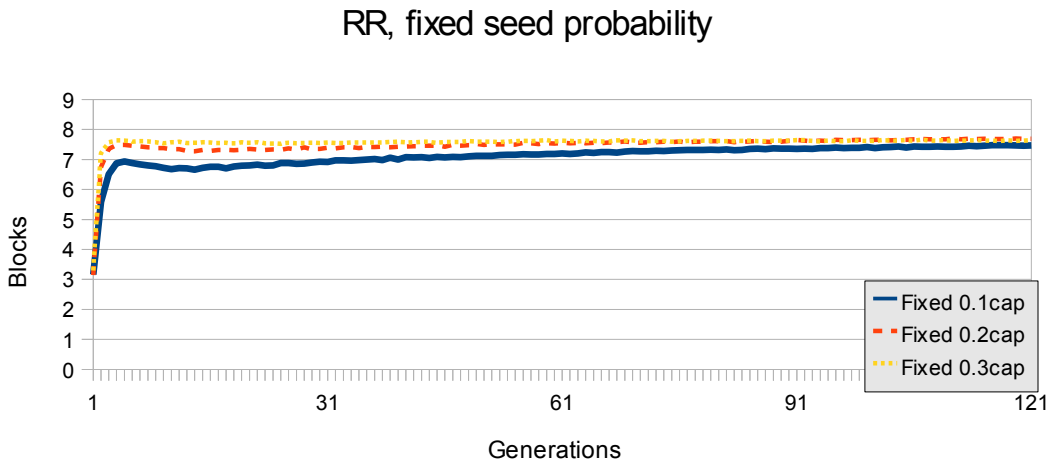

Figure 4.1.b: Mean number of blocks present during the first 120 generations of 1000 runs of the SGA over the RR problem, using 2-point crossover, 5-tournament, and *flat* seed probabilities.

From figure 4.1.b, we can see that the static probability function finds, on average, seven blocks within the first three generations, and then peaks below eight after another three generations. By contrast, the fast and slow linear increase functions take 15 and 90 generations respectively to reach seven blocks (figure 4.1.c, figure 4.1.d). The 0.1 and 0.075 log increase functions take 25 and 30 generations respectively (figure 4.1.e, figure 4.1.f). For each of these dynamic seeding probability functions, we observe a small initial *drop* in block saturation. This corresponds to the zero seeding probability at

11 The performance for the cap 0.2 was better than for either of the other tested caps, for all functions.

generation zero coupled with the extinction of randomly generated blocks in the first new generation through mutation and non-selection.

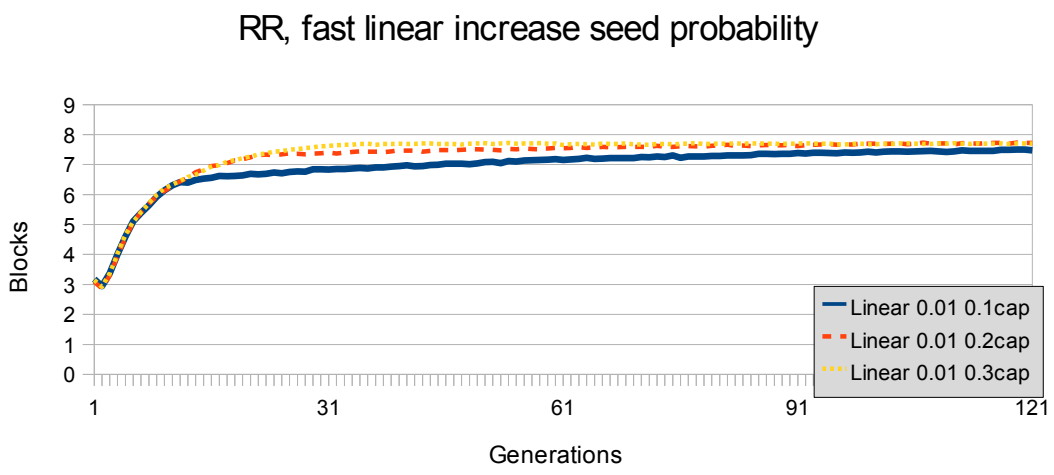

Figure 4.1.c: Mean number of blocks present during the first 120 generations of 1000 runs of the SGA over the RR problem, using 2-point crossover, 5-tournament, and seed probabilities generated by a *fast linear increase* function.

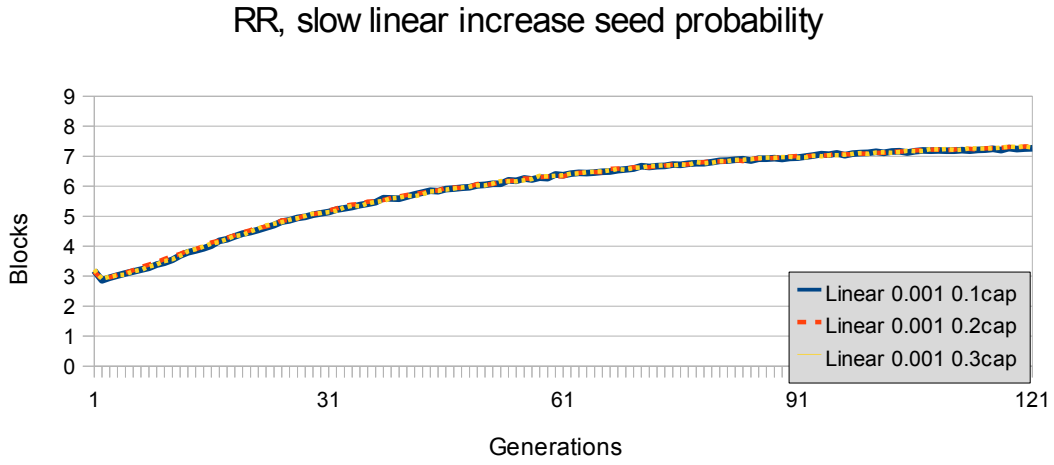

Figure 4.1.d: Mean number of blocks present during the first 120 generations of 1000 runs of the SGA over the RR problem, using 2-point crossover, 5-tournament, and seed probabilities generated by a *slow linear increase* function.

It is apparent, from the table and figures, that 'increasing' seed probability functions simply lose the benefit of seeding in the early stages of the algorithm, without any real gains later in the run. We take this as a strong indicator that the simple flat seed probability function is at the very least *comparable* to other plausible functions, and therefore continue to use it throughout this work.

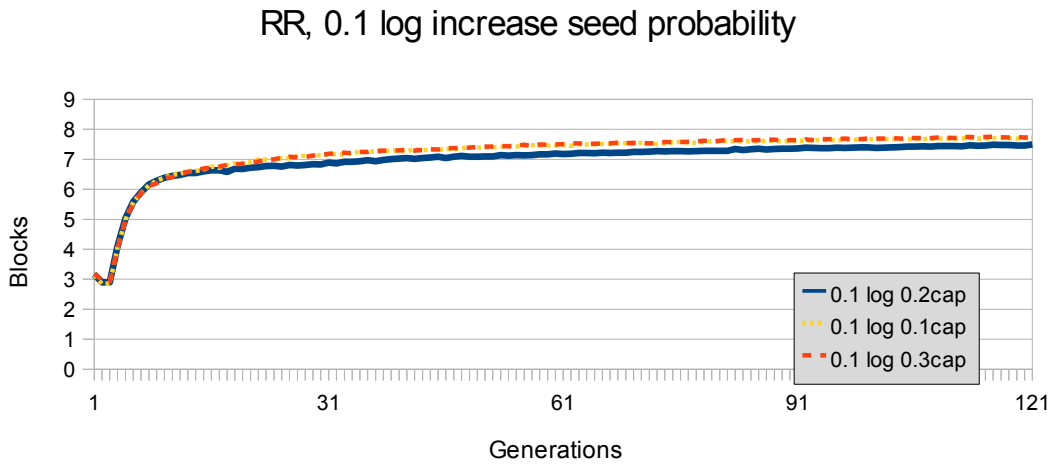

Figure 4.1.e: Mean number of blocks present during the first 120 generations of 1000 runs of the SGA over the RR problem, using 2-point crossover, 5-tournament, and seed probabilities generated by a *0.1 log increase* function.

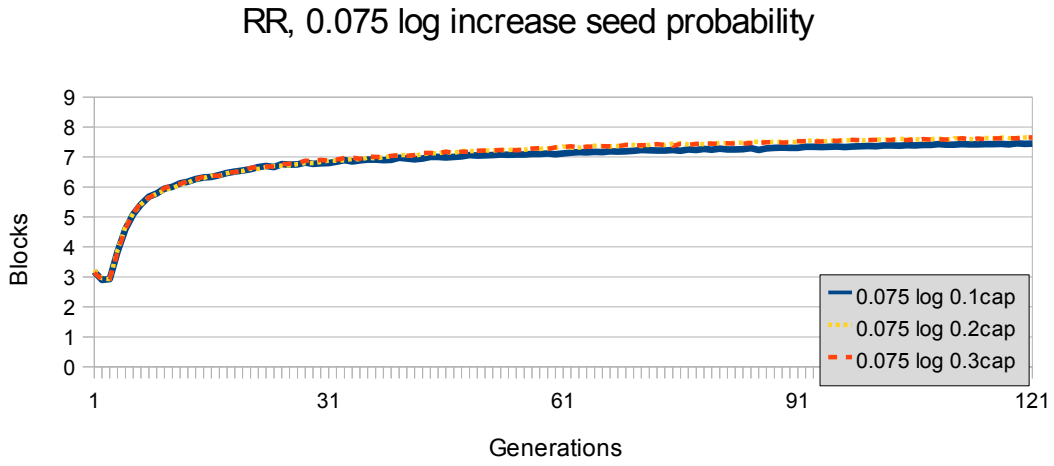

Figure 4.1.f: Mean number of blocks present during the first 120 generations of 1000 runs of the SGA over the RR problem, using 2-point crossover, 5-tournament, and seed probabilities generated by a *0.075 log increase* function.

[4.2] Seeding with Random Individuals

The results of section 4.1 appear to bear out Skinner's [2009] assertion that the usefulness of seeding lies in the introduction of *superior* genetic material (building blocks). However, we still wish to know exactly how much benefit seeding provides in the form of preventing early convergence by introducing 'fresh' genetic material.

We separate this from the utility of building blocks as follows. We compare the CGA to a form of the SGA which seeds with randomly generated individuals. This is accomplished simply by initialising the seeding algorithm with a presample size of 50 and seed pool size of 50. The seed individuals have

therefore not been subject to selection at all. The test was run on the RR problem with 5-tournament, 2-point crossover, and other standard parameters.

Essentially, what we are comparing is the power of the CGA's mutation operator against the power of the SGA to prevent early convergence, in each case by introducing random material. Of course, there is a chance that the mutation operator might introduce a new block, but the analysis in [Skinner, 2009, pp 54-65] says this is remote: for the RR problem at least, the vast majority of the work of discovery is done during recombination. Mutation has an expected discovery rate of <0.001.

Similarly, seeding with random individuals could technically introduce new blocks, but this is again unlikely, as the chance that any random individual contains any block at all is 0.031 (see Appendix C, section C.2.1), and the more blocks there are in the main population, the less chance the offspring of a seed individual will be fit enough to survive.

	Seed probability					
	0 (CGA)	**0.1**	**0.15**	**0.2**	**0.25**	**0.3**
Generations	257.3 (134.5)	*600.0 (246.1)*	*555.7 (249.7)*	*555.9 (237.5)*	*587.5 (225.9)*	685.0 (288.7)
Number completing	1000	430	606	676	398	3

Table 4.2.a: Mean number of generations to complete and number of runs completing over 1000 runs of the SGA on the RR problem with random seeding, compared to the CGA. Standard deviations are given in parentheses (σ). Results statistically significantly different from the CGA are given in *italics*.

The results are given in table 4.2.a. The CGA significantly outperformed the SGA (with 95% confidence) in almost every instance[12]. It appears that seeding with random individuals is harmful, and thus most of the benefit of the seed pool must come from the fact that it is *building blocks*, not just convergence-preventing random variation, being introduced by seeding.

Note, however, that *some* (0.15 - 0.2) seeding with unfit individuals is (statistically significantly) better than *minimal* (0.1) seeding. It seems likely that the seeding function is acting as a crude macro-mutation operator, in the absence of the regular bit-mutation of the canonical GA, and is thereby in fact approximately simulating the functioning of the CGA.

[4.3] Initial Conclusions

Skinner's results are, broadly speaking, repeatable. Seeding so far has proven robust and effective. It is a considerably different type of tool than the other types of GA-variant designed to prevent early convergence which we have encountered.

12 It also obviously outperformed the SGA with 0.3 seed probability, intuitively speaking, but the usual Student's *t*-test is not suitable for the tiny fraction of a distribution for which we have data. We instead used the Welch form of the *t*-test, but the P value calculated was not less than 0.05.

Seeding is doing more than simply enforcing diversity, but providing the building blocks upon which the algorithm depends. Its ability to do so through little more than *initial random search* has some rather worrying implications for the canonical genetic algorithm.

In order to determine exactly how well the seeding algorithm can perform, we spend the next two sections investigating its various parameters. We wish to find optimal seed pool sizes and seeding probabilities, and discover how to most efficiently and effectively fill the seed pool.

[5.] Determining Parameters: Presample Size and Seed Probability

[5.1] Introduction

[Skinner, 2009] used a presample size of 1000 throughout. We conducted experiments varying the size of the presample used from as little as 500 to as much as 750000, typically over around 20 categories for each set of conditions.

The presample size we tested for each experiment was tweaked over its duration, to maximise performance. For example, if it was clear from an early sample of presample sizes[13] that the algorithm was not going to give any results, we did not try all the intermediate values. Conversely, if all the best results were found between two presample sizes, we introduced more categories between those sizes in order to pinpoint the best range.

For each presample size, five different seeding probabilities were tested: 0.1, 0.15, 0.2, 0.25, and 0.3. The standard parameters, including 5-tournament, 0 mutation rate, and seed pool size $n = 50$, were used. We separately tested uniform and 2-point crossover for each of the six problems.

Therefore the experiments in this section range over four variables: problem type, crossover type, seed probability and presample size.

We *normalised* the data by calculating the cost of seed pool generation (assuming the computational bottleneck is fitness evaluation). See Appendix B for an explanation.

The *full results* for these experiments are given in Appendix C, for reasons of space. We summarise those results in section 5.2 below, and draw more general conclusions in section 7.

13 Such as {1000, 10000, 50000, 100000, 500000}

[5.2] Results

[5.2.1] Royal Road (RR)

We first examined the Royal Road problem. The CGA completed this problem with 2-point crossover in a mean 256.6 generations. In the experiments, the seeding algorithm was able to reduce this by as much as 233 generations (a 90% reduction), or by 208 generations (an 80% reduction) after normalisation.

After normalisation, the optimal values were a presample size of 80000 – 100000, and a seed probability of 20%. The seeding algorithm still significantly outperformed the CGA with a seed probability as low as 0.1 or as high as 0.3.

In terms of the raw mean generations the SGA takes to finish, we run into diminishing returns as we start increasing the presample size beyond 150000. In fact, there is no statistically significant difference between a presample of 200000 and a presample of 600000 for any of the seed probabilities. By contrast, a presample increase just *one quarter* of that size, from 50000 to 150000, improves our mean generations to complete from 17.8 and 47.9 generations.

Numerical analysis (given in Appendix C, section C.2) suggests that a presample of 90000 would have an average of 1.7 individuals with *at least three* correct blocks each. So it would be usual for at least one individual in the seed pool to have at least three blocks, for what we have determined to be the "optimal" presample size. The fact that finding 3/8 of the solution to the RR problem by what amounts to *random generation* is vastly more efficient than running the CGA from scratch could be seen as either a spectacular failure on the part of the CGA, or great promise on the part of the SGA.

We repeated these experiments with uniform crossover, and found that the SGA performed more poorly than the CGA. The ability of the CGA to perform discovery through the bit-mutation operator is apparently greater than the ability of the SGA to perform discovery through massive disruptive crossover of potentially fit seed individuals. In either case, the probability of discovering a block via uniform crossover is analogous to doing so through eight random mutations, and the probability of *retaining* a new block being introduced by a seed individual is almost as low.

For the uniform crossover variant of the SGA, the optimal parameters were a presample size of 100000 – 200000, with a seed probability of 0.15 (after normalisation). However, few runs (seldom more than a quarter) actually completed.

The success of the CGA over the SGA for RR with uniform crossover compelled us to investigated a combined form: using the SGA with a 1/128 bit-mutation rate (half the rate of the CGA). Any improvement over the strict seeding algorithm – and any reduction in the optimal seed probability – could be taken as support for the hypothesis that the poor performance of the SGA is due to the superiority, for block discovery and retention, of the CGA's micro-mutation to the SGA's limited ability to perform macro-mutation (through crossover and seeding). In fact, however, there was little improvement.

[5.2.2] Staggered Royal Road (R2)

The seeding algorithm was barely able to solve the R2 problem at all, with fewer than half the runs completing with a presample size of 500000. In comparison, the CGA completed this problem using a mean 277.29 generations.

The poor performance of the SGA is not surprising. We used a 2-point crossover operator, which is likely to be destructive overall (given the staggered nature of the problem). According to Skinner [2009, p 68], "we would expect that if we use R2 as our test problem that 2-point crossover should fail because it simply is not capable of combing any two blocks without also disrupting every other block."

Why was the CGA able to solve the problem? Very likely due to bit-mutation. When two parents A and B reproduce via 2-point crossover, the offspring will only contain a block present in parent A if either

(a) the block is also present in the parent B, or

(b) parent B happens to have the corresponding part of that block (an average of 4 bits) correct, through random chance.

Thus the discovery rates through crossover *or seeding* are going to be extremely low.

Consider the case where 2-point crossover brings together several sections of a genome so as to almost form a block, except for a single bit at any arbitrary position within that block. The CGA has a 1/64, or 0.01563, chance of then successfully completing the block through mutation. Although it does then have a $(1 – (63/64)^7)$, or 0.10438, probability of destroying the new block through mutation, the chance of 2-point crossover bringing together 7/8 of any block, for a string of 8 staggered blocks, is actually *relatively* high when mutation is being performed. The SGA lacks this ability.

Uniform crossover, although "jumbling" the genetic information, should still be much more capable than 2-point crossover of successfully crossing two staggered blocks. We therefore expected the SGA to perform much better on the R2 problem using uniform crossover than it did with 2-point crossover.

The CGA completed the uniform variant using a mean 237.1 generations (slightly better than using 2-point crossover). While the SGA performed better than it did on the 2-point crossover version, it still failed to match the performance of the CGA, even before normalisation for the cost of the seed pool. After normalisation, its best performance was 315.0 generations, at a presample of 200000 and seed probability 0.15 – essentially the same as for the RR problem using uniform crossover. Of 1000 runs, 987 solved the problem with these parameters.

[5.2.3] Hierarchical Royal Road (HRR)

The CGA completed this problem with 2-point crossover in a mean 268.3 generations. In our experiments, the seeding algorithm was able to reduce this by as much as 233 generations (a reduction of 94%), or by 220 generations (a reduction of 80%) after normalisation. This is an excellent result, comparable to or even better than the non-hierarchical version. It is made even better because the Hierarchical Royal Road has been noted to be an "unusually difficult" problem for the CGA [Mitchell *et*

al, 1992]. Hitchhiking[14] appears to not be as significant a problem for the SGA as it is for the CGA.

The optimal normalised values were a presample size of 80000 – 120000, with a seed probability of 0.2 or slightly lower. The SGA's performance on HRR continued to improve up until the largest presample size tested (unlike for RR). This is because at the scale we tested, the probability of improving performance by getting *one more block* in a seed individual is negligible from the fourth block, but improvement by getting *more contiguous blocks* is still well within the realm of possibility.

As with the RR and R2 problems, the SGA performed poorly on the HRR problem with uniform crossover, where the CGA was able to achieve decent results.

Some of the results with uniform crossover were rather incongruous; a *decrease* in performance began to emerge at the largest presample sizes tested. We used further experiments to pin down the reason for this anomaly.

The SGA with uniform crossover does better on the HRR problem when its seed pool is filled with individuals containing two primary blocks than it does when its seed pool is filled with individuals containing a single secondary block.[15] Given that a smaller presample tends to fill the seed pool with the former type of individual, which are 'crowded out' with the latter type as the presample gets larger, this preference leads to the strange behaviour observed.

Further analysis shows that the correlation between *"number of generations required to solve"* and *"fitness of seed pool individuals according to HRR fitness function"* is considerably lower than the correlation between *"number of generations required to solve"* and *"number of bits set to 1 in seed individuals"*. What this means is that the SGA will choose individuals with a secondary block from the presample to populate the seed pool, because they have a higher fitness according to the HRR fitness function, even though the SGA performs better using those (less-fit) individuals with several non-adjacent primary blocks. This adds up to the fact that the hierarchical fitness function is simply flawed.

See Appendix B for a more extensive discussion of these anomalous results and the extra experiments we undertook.

[5.2.4] One-Sum (OS)

The CGA was able to complete this problem in a mean 18.2 generations with 2-point crossover. In our experiments, the seeding algorithm was only able to beat this performance with the very highest presample sizes, and did not even match the result after normalisation.

From a normalised perspective, the optimal parameters were a presample size of 10000 – 20000, with a seed probability of 0.1 – 0.5. However, the best results were not tightly bound to a particular seed probability or presample size – perhaps due to the 'easy' nature of the problem.

14 We describe this issue briefly in section 2.4.
15 This is very likely due to the hitchhiking effect.

Note that, because there are no building blocks to find, the main population should have reached the fitness of the seed pool within just a few generations, and after that point, seeding may in fact begin to hinder the process. We conclude that it is important, when using the SGA, to have a large enough presample and low enough seed probability to make sure the seeding is not *harming* the main population, but seeding *per se* is not very helpful on this problem.

The variant using uniform crossover is much easier for the CGA to solve: its performance improves from a mean 18.2 generations to 12.2 generations. The lack of building blocks means the uniform operator cannot cause disruption. Using this crossover operator made little difference to the SGA.

[5.2.5] Hierarchical If-And-Only-If (HIFF)

On the difficult and local-maxima-dense HIFF problem, which the CGA was essentially unable to solve, the SGA performs reasonably well – finding a solution 90% of the time with sufficiently large presample sizes. With uniform crossover, the SGA is barely effective, only stumbling on the solution 1% of the time – little better than the CGA.

Uniquely of our tests so far, the SGA performed best with high seed probabilities. This may be because bit-mutation will essentially *never* free it from a (possibly quite extreme) local maximum. Conversely, as long as seed individuals are introduced into the main population in sufficiently large numbers, it should be possible to either combine those individuals into partial solutions rivalling the fitness of any partial solution the main population has found, or combine those individuals with the partial solution of the main population to improve it.

We ran extra experiments up to 20000 generations (a twentyfold increase), with 100 runs each. We focused on the higher seed probabilities which the SGA performed better on: 0.2, 0.25, 0.3 and 0.35. With a presample of 400000, the SGA almost always solved the HIFF problem, except with the lowest tested seed probability. The number of generations it required at that point was 500 to 700 for optimal seed probabilities of 0.25 to 0.3, with 2800 generations for 0.2 seed probability and 1800 generations for 0.35 seed probability.

The optimal presample size after normalisation appears to be 500000. For presamples of at least 10000, we can say with statistical confidence that the SGA outperformed the CGA even with less-suitable seeding probabilities.

[5.2.6] Deceptive Trap (DT)

The CGA is *unable* to solve the deceptive DT problem: it did not have a single successful run from the fifteen thousand or more that we ran. Any result at all from the SGA could therefore be seen as an improvement. In fact, the seeding algorithm performs rather well: better, in fact, on DT than it does on HIFF.

A seed probability of 0.15 to 0.2, and a relatively large presample of 100000 to 300000, appears to be optimal after normalisation. The SGA is sufficiently powerful to solve the DT problem 100% of the time for a presample of 500000; this is the computational equivalent of running the CGA for another 200 generations.

The CGA struggles to find each block, due to deception. The SGA only needs that block to be found once in the entire presample for it to appear in the seed pool and be introduced into the main population through 2-point crossover.

As might be expected, the SGA cannot solve the DT problem using uniform crossover: the disruption combined with deceptiveness is an insurmountable problem. Just as for the canonical algorithm, not a single run completed.

As with the HIFF problem, we ran extra experiments up to 20000 generations, with 100 runs each. We focused on the lower seed probabilities. By the time we reached a presample as low as 25000, the SGA was finding the solution 100% of the time, with all seed probabilities. There was no specific combination of seed probability and presample size that worked particularly well; instead the SGA performed generally well on presamples of 100000 to 500000 and 0.1 to 0.2 seed probability.

Remarkably, even when a seed pool of 50 was drawn from a presample of 100, two-thirds of the runs were able to solve the DT problem within 20000 generations. Such a presample would have an average of 32 individuals with one or more blocks correct, but these would not necessarily be chosen for the seed pool: they could be outweighed by individuals with no blocks completely correct, but multiple blocks in high-fitness (and therefore highly deceptive) states.

[6.] Determining Parameters (II): Presample Size and Seed Pool Size

[6.1] Introduction

We now have a reasonable idea about the appropriate seed probability, both in general and for each of the six problems in our problem set. We turn our attention to trends in the optimal seed pool size for various presample sizes.

We vary the the size of the seed pool from $n = 5$ to $n = 5000$. The seed probability for each experiment is fixed at its previously discovered optimum level. For each seed pool size, different presample sizes are tested. We use 5-tournament and other standard parameters, and compare uniform and 2-point crossover.

We expected one of two possible outcomes for these experiments: either

(a) a pattern specific to each experiment whereby the best results would come from having the seed pool size as a particular *proportion* of the presample[16], or

(b) a fixed optimal seed pool size for classes of problem, given some minimum presample size.

We consider the former to be more likely than the latter: even a very large presample is going to populate the seed pool with dross if the seed pool is – for example – half its size. However, a very small seed pool size is not necessarily the answer, as we want many *different* blocks to be present in our seed pool.

We again *normalised* the data by calculating the cost of seed pool generation (assuming the computational bottleneck is fitness evaluation); see Appendix B.

The *full results* for these experiments are given in Appendix D, for reasons of space. We summarise those results in section 6.2 below, and draw more general conclusions in section 7.

[6.2] Results

[6.2.1] Royal Road (RR)

With a seed probability of 0.2, it seems that our initial seed pool choice of size 50, taken from [Skinner, 2009, pp 92-93], was a good choice, as a seed pool of 50 to 100 appears to be optimal. The *viable* seed pool size increases with the presample size, as expected, but the *optimal* size is fairly consistent.

The SGA was able to improve on the performance of the CGA (256.6 generations) with a seed pool as small as 25. This improvement was statistically significant for each presample size tested.

When we tested uniform crossover (using a seeding probability of 0.15), we found the SGA was still able to solve this problem/parameter combination most of the time. However, the CGA outperforms the SGA for the full range of seed pool sizes we tested, as expected (see section 5.2.1).

[6.2.2] Staggered Royal Road (R2)

With an optimal seed probability of 0.2, the SGA performed poorly (as we saw in section 5.2.2). It appears that the seed pool size begins to act to the detriment of the algorithm when it increases beyond 100, but we cannot say so with statistical rigour when so few runs completed.

Uniform crossover was slightly more effective. Using a seed probability of 0.15, the SGA was at least able to solve the problem most of the time for sufficiently large presample sizes. Once again, a seed pool of 50 individuals (or slightly larger) appears to be optimal for these circumstances.

16 This must be differentiated from the strict, presumably universal benefit of using a larger presample.

[6.2.3] Hierarchical Royal Road (HRR)

This is one of the problems which we have observed the SGA does well on. We used seed probability 0.2, and found a smaller seed pool size of 25 to 50 gave the best performance.

Recall that HRR with uniform crossover is the problem/parameter combination with the unusual behaviour, as discussed in Appendix E. We tested the problem with seed probability 0.15, and found that at the upper reaches of the presample size, where the negative influence begins to show, the larger seed pool – as large as 500 individuals – performs better.

This is exactly as expected if our earlier hypothesis is correct: a sufficient number of individuals with two non-contiguous blocks or even three blocks are getting into a seed pool of size 500. These individuals are being crowded out in a seed pool of size 50 by individuals with a single secondary block, which the HRR algorithm classifies as "more fit" even though they are actually less useful for the progress of the genetic algorithm.

[6.2.4] One-Sum (OS)

For the OS problem with a seeding probability of 0.15, we found that a seed pool of approximately 50 was the "magic number". The optimal presample size, after normalisation, for this problem is quite small: in the 5000-10000 range. This is likely because the algorithm is able to build up more fitness for this simple problem in a few generations than it is by increasing the presample by the amount with a cost corresponding to that number of generations.

We also tested uniform crossover, with a seed probability 0.1, and found a small presample of 1000-10000 worked well. The problem was readily solved with seed pools as small as 5 or as large as 500.

[6.2.5] Hierarchical If-And-Only-If (HIFF)

Recall that the CGA was essentially unable to solve the HIFF problem. Using a seed probability of 0.3, we found that the seeding algorithm was most effective with a seed pool size around 50, or slightly larger with the largest tested presample. It was only at this level (500000 individuals presampled) that the SGA managed to complete 90% of the time.

[6.2.6] Deceptive Trap (DT)

The CGA fails on the DT problem, and the SGA is rather successful. We used a seed probability of 0.175 for our experiments, and found a trend towards a reasonably large optimal seed pool size (50 to 100). As we increase the presample size, the first seed pool size on which all runs completed was 1000; the second was 500. A seed pool of 100 is statistically significantly better than 50 for presample size 100000, and is significantly better than either 500 or 25 for presample sizes of 5000 and higher.

The apparent usefulness of larger seed pools may be due to the smaller block length in this problem: the ratio of seed-pool-size-to-presample-size can become narrower before the seed pool begins to fill with 'garbage' individuals containing few or no blocks.

[7.] Conclusions

[7.1] Analysis of Derived Parameters

We have discovered the optimal or near-optimal parameters for our test problems. We can make observations about the similarities and differences between the best parameterisations of the problems we tested, and make several inferences about them.

Consider first the Royal Road problems. RR and HRR are very similar. They both have ideal presamples of 100000 or just below, seed probability of 0.2, seed pool of 25 – 50 individuals, and reduce the running time of the CGA by around 80%. The *spread* of parameters on which they do reasonably well is also very similar – compare tables C.2.1.b and C.4.1.b in Appendix C, for instance.

The addition of the 'hierarchical' part of the problem only really affects the point at which we encounter diminishing returns[17] and the effects of uniform crossover. The SGA does not work with uniform crossover on HRR, whereas it does on RR, encountering some incongruous results as explained in Appendix E.

Performance on the staggered variant, R2, is quite dissimilar. This is not surprising. The SGA performed very poorly, because 2-point crossover is an inappropriate operator for dealing with maximally disparate blocks. Only half the problems completed; this was considerably improved by using uniform crossover. R2 was the *only* problem which the SGA did not perform well upon.

OS, as our simple 'sanity check' problem, was easily solved with a range of parameters for the SGA. The algorithm did not seem to 'care' whether the seed pool contained 25 or 100 individuals, or whether the seed probability was 0.1 or 0.25. The gains of a larger presample were offset by the improvement the canonical algorithm could make by normal running through the equivalent generations, giving the OS problem our lowest optimal presample size of 10000 – 20000.

HIFF, conversely, had the highest optimal presample size of 500000. It also had the highest seed probability of 0.25 to 0.3. We suggest that this is due to the fact that the problem's fitness landscape is filled with local maxima, so seeding – specifically, seeding with *good* individuals – is commonly in demand to 'knock' it out of a local optimum. The set of parameters the SGA did well on for the HIFF problem was the narrowest of all our problems (see table C.1.9.b in Appendix C).

17 RR stops noticeably improving with presample sizes beyond 200000; HRR continues to improve.

Interestingly, despite the deceptive nature of the DT problem, the SGA solved it more readily than it did HIFF. It was solvable with a range of parameters, including an optimal presample of 100000 to 500000, and a rather low seed probability of 0.1 to 0.2. This low seed probability is interesting, given that the genetic algorithm without seeding (the CGA) utterly fails on the DT problem; this may be a topic for further enquiry.

The fact that our implementation of Deceptive Trap had a bit-string of length 60, compared to the length 64 for all other problems, may have played some part in the good performance of the algorithm. It may also be that the *large* number of *small*, *discrete* blocks[18] is well-suited to the genetic algorithm, once the problem of deception is mitigated by seeding.

The SGA with uniform crossover did poorly on most of our problems – on RR and HRR it works sometimes, and on HIFF and DT not at all. This is not a surprise, given the fundamentally block-disrupting nature of the uniform crossover operator. As the OS problem does not have multi-bit blocks to disrupt, the success of uniform crossover there is completely expected.

For all the problems, Skinner [2009, pp 92-93] appears to have hit the nail on the head with a seed pool size of 50. OS was generally easy, working with a range of seed pool sizes. R2 and DT also did well with a larger seed pool; the DT problem has twelve different well-defined blocks, so a larger seed pool might usefully carry more of the required genetic variation. R2 was more of a 'problem case', and it is difficult to draw conclusions for it when so few runs completed.

RR, HRR and HIFF also did well with a smaller seed pool. When there are only eight (lowest-level) blocks, it is not surprising that RR and HRR could still get the necessary variation from 25 seed individuals.[19] The lower optimal seed pool for HIFF is something more of a mystery. HIFF has, technically, the highest number of different blocks of all the problems we tested. However, many of them are contradictory, and *all* can lead to local optima depending on the current state of the main population. Further research is required here.

We found Skinner's seed probability of 1/3 to be too high, except in the case of HIFF. The other problems tended to prefer a seed probability of 0.2, and were sufficiently sensitive to typically do poorly on a 0.3 seed probability.

From section 6, we conclude that the ideal seed pool size is not a simple fixed fraction of the presample size. We confirmed that

(a) it was important to perform seeding with high-quality individuals, and therefore

(b) it was important to have a high enough ratio of presample-individuals-to-seed-pool-individuals to ensure we fill the pool with useful genetic material.

However, there are also other factors modulating it. That a larger presample is always better (before normalisation) is obvious. There also seems to be a problem-dependent factor.

18 The DT problem has 12 blocks. The HIFF problem technically has 254, but they are massively interdependent. Similarly, the HRR problem has 14 blocks, again interdependent. RR has only eight blocks.

19 And as we show in Appendix E, it is these lowest-level blocks which hold the most weight for HRR.

[7.2] Evaluation of the Seeding Algorithm

In Appendix E, we noted that the SGA cannot compensate for an inappropriate operator or a poor fitness function. However, in general, how good is the SGA? If the SGA is to be used in a real-world domain, we cannot initialise it with the optimal parameters, since these are almost certainly not known in advance. So, how close do they have to be to the optimum for the SGA to to still work? How easy are they to guess?

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, and 50 seed pool individuals drawn from a presample of 100000 will work considerably better than the CGA on *any* of our problems, other than OS (which is too easy) or R2 (which is designed to be problematic for our crossover operator).

We do not claim that these 'optimal' and 'appropriate' parameters are universal. They may not necessarily carry over to wider domains, real-world problems, or more complex building block situations... but we are hopeful.

The genetic algorithm already has a reputation for being sensitive to small changes, and we have added several more parameters to it. With a sufficiently large presample the SGA works, and a seed pool of 50 appears to be fairly constantly appropriate, but the seeding probability may require fine-tuning on a case-by-case basis. In some domains it may be possible to run “exploratory tests” to narrow down an appropriate seed probability.

What of the question: when is it *appropriate* to apply the SGA? We have seen that it does not work (with the standard 2-point crossover) on a maximally staggered building block problem. Such artificial problems, designed to be GA-hard under the building block hypothesis, are unlikely to occur in other domains, but some staggering is possible. We did not test intermediate, or partially staggered, problems. We would hypothesise that the SGA becomes progressively worse as 2-point crossover becomes a less appropriate operator.

We don't typically know the overall composition of a given problem, of course, so it is hard to make strong claims about the suitability of the SGA. We will note, though, that wherever the CGA works (with *n*-point crossover), the SGA should.

We believe we have appropriately extended the work of [Skinner, 2009]. Although our results are only an approximate numerical match (see section 3.3) and his work inexplicably succeeds with a 1/3 seeding probability, the general trends observed are the same. Having established a baseline, we feel we have succeeded in our goals of obtaining a set of parameters for the SGA, establishing where it does and does not work, and discerning the reasons behind some unusual results. In the process, we have run more than *1300* experiments, involving over *1200000* runs of the genetic algorithm.

We consider this to be a good result: not only an 80% reduction in the difficulty of problems the GA struggles with, such as the Royal Road, but also the overcoming of massive local optima in the case of the Hierarchical If-And-Only-If problem, and deceptiveness in the case of the Deceptive Trap, which the CGA is not equipped at all to deal with.

There have been numerous extensions to the "canonical" GA, many of which address the same failings we have overcome here. The utility of the seeding operator may lie in its broad range of effects: preventing early convergence *and* overcoming local optima *and* overcoming partially deceptive problems.

Note that, for a seed probability p and crossover probability q, with two children per crossover event, 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, i.e.

$$2p^2 + 2p(1-p)q + p(1-p)(1-q)$$

For example, for our typical seed probability 0.2 and crossover probability 0.9, the fraction of individuals in the next generation containing new seeded material is 0.08 + 0.288 + 0.016 = 0.384. This will be the case in generation 1 and in generation 1000.

With this in mind, we submit that many of the other GA extensions in the literature that are designed to prevent early convergence, such as geographically distributed populations or the island model, are not as capable as the SGA of maintaining diversity – serving to *slow down* or *discourage* early convergence. When a third of each generation of the SGA contains 'fresh' seed pool material, regardless of the fitness of individuals in the main population, it is difficult to see how the population could converge to anything but the solution.

[7.3] Further Research

Comparisons to the 'state of the art' would be viable directions for future research. We believe we have demonstrated that the SGA is an improvement over the basic CGA, but as we have already pointed out, the canonical algorithm is seldom in use – presumably *because* of some of these flaws which [Skinner, 2009] aimed to fix.

To that end, it might be interesting to *combine* the SGA with other common GA modifications, to see what best achieves our aims of a robust, versatile "out of the box" GA solution. Viable candidates for combination might be: adaptive variation of parameters, dynamic variable encoding, or 'messy GAs'. We could even test the performance in combination with local search or simple hill-climbing.

There are also variants like probabilistic model-building genetic algorithms (PMBGAs), sometimes called estimation of distribution algorithms [Mühlenbein & Paab, 1996]. The modern Bayesian Optimisation Algorithm (BOA) and its Hierarchical variant [Pelikan & Goldberg, 2000] are a similar case... although Skinner [2009, p 42] points out that BOA fails on the Royal Road problem with a population size of 128 (100 runs), and only solves it 82% of the time with a population of 1000. Regardless, some versions are better at some aspects of the wide domain of *GA problems* than others.

One promising variant for a hybrid SGA would be a GA dynamically re-mapping its own genome, as described in [Sehitoglu & Ucoluk, 2001]. Such a method is able to reorder building blocks, evolving towards permutations of the chromosome encoding that spatially re-clusters relevant genetic information in the genome. This has been observed to both converge faster and better avoid local minima. [Sehitoglu & Ucoluk, 2001] This re-mapping towards non-staggered blocks should allow the SGA to solve problems

like R2 – *the one weakness we were able to discover* – with relative ease.

Further work might also explore the quirks of each particular problem in more detail. For instance, we found that the CGA only worked on HIFF with rank selection and 2-tournament selection. It might be illuminating to run experiments with the SGA using different forms of selection.

In section 4.1 we explored a few non-static functions for seeding. These – and other functions – should be tested in other problem spaces. Also, other variations of the way in which seeding happens could lead to further refinement of the SGA. In the best case, it may be possible to deduce, through measurement, the relative merits of seeding at each point in the algorithm. This would allow us to dynamically 'throttle' the seeding probability to an appropriate level.

We did not test different ways for the seeding operator to affect the main population. It would be interesting to see the effects of generating the first generation of the primary population entirely from seed pool individuals. Alternatively, we could inject seed individuals into the population only when the offspring and/or parents of some crossover event fall below some threshold. There are many more avenues to pursue here.

We also did not focus as extensively on rates of *discovery*, *combination* and *destruction* as Skinner [2009] did. Tracking these more closely throughout the experimental process might allow us to be more precise about what constitutes an optimal parameter value for the SGA.

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.

[8.] Appendices

Appendix A: Statistical Models and Tests

In this dissertation we use two different tests for statistical significance. The nature of our data is often suitable for the standard Student's *t*-test: large sample sizes following a Gaussian distribution. The random element in the initial conditions for each run of the algorithm[20] making paired tests unsuitable. For some of the P values in this work, we therefore use a two-tailed unpaired Student's *t*-test.

However, we have a problem: "censored" data. Whenever a run does not complete within the time bound (variously, 100 or 1000 or 20000 generations), we note it down as non-completing and move on. What do we do with these data points when we want to make statistical inferences?

The extreme approach of discarding the censored data (assigning it a zero weight, so that means and standard deviations are calculated without it) is fundamentally flawed: [Etzioni & Etzioni, 1994] demonstrate that we *cannot* validly extrapolate from censored data. It is unlikely that the remainder of our data follows a standard Gaussian distribution, so a test like the *t*-test which presumes a normal distribution is inappropriate.

[Etzioni & Etzioni, 1994] give a *low-power* test for statistical significance with a strong *guarantee*; namely, that the test computes an absolute upper bound on the P value that would be calculated if the experiment were run without a time bound. This can be guaranteed because their tests take every instance of doubly-censored data as support for the null hypothesis. However, this test presupposes paired data. Each run of our algorithms starts from random initial populations and presamples, so it would not be entirely appropriate to pair runs across experiments.

We turn, therefore, to the *bootstrap test* described in [Cohen & Kim, 1993]. This is a two-sample test, finding the difference between the mean times of two algorithms A and B. The test establishes whether that difference is significantly different from the zero value that the null hypothesis predicts.

This is done by drawing many "bootstrap samples" from the combined results of A and B, two at a time, and establishing the difference of the sample means, T, for each pair. Under the null hypothesis, A and B are not different, so we can establish a sampling distribution from the T values which should be centred around zero. Let X be the difference of means of the uncensored values in the results of A and B. Our P value is then simply number of values in the T distribution that are less than or equal to X, divided by the size of the T distribution.

We use the bootstrap test for all cases where we have censored data. [Etzioni & Etzioni, 1994] point out that the term *significant difference* refers to statistical significance, not to the magnitude of a difference; even a tiny difference may turn out to be statistically significant with large enough sample sizes. Therefore in this work, we only consider the question of significant difference when we already consider the observed difference to be non-trivial.

---

20 For example, the randomised presample and randomised initial population.

Appendix B: a Note on Data Normalisation

Our motive for data normalisation is simple: increasing the presample size for the seeding algorithm results in better and better 'seed individuals'. We need to consider the costs associated with seed pool generation, otherwise our 'ideal' presample would tend towards whatever enormous size actually randomly generates the solution. We need to be able to relate the cost of a certain presample to the cost of another, and to the cost of the CGA.

We can calculate the cost of seed pool generation by assuming that the computational bottleneck is the number of fitness evaluations performed. We follow the reasoning of [Skinner, 2009, p 93]. When the canonical algorithm (CGA) is run, we can calculate how many fitness calculations N it makes per generation, based on the population size, selection type and number of children generated per crossover event. Since one fitness calculation per individual must be made to construct the seed pool from the presample, it follows that a presample size of N is equivalent to running one extra generation of the CGA. Dividing a presample by N thus gives us the additional cost of the seed pool generation.

This is a rudimentary estimation, ignoring overhead costs such as those involved in creating and sorting n random individuals, but it captures the main issue of scale.

Appendix C: Determining Optimal Presample Size and Seed Probability

We performed experiments by varying the presample size from as small as 500 to as large as 750000. For each presample size, five different seeding probabilities were tested: 0.1, 0.15, 0.2, 0.25, and 0.3. The standard parameters, including 5-tournament, 0 mutation rate, and seed pool size $n = 50$, were used. We separately tested uniform and 2-point crossover for each of the six problems. Therefore the experiments range over four variables: problem type, crossover type, seed probability and presample size.

For each experiment, we present the results first in a table with the associated standard deviations, and then as a table of *normalised* data, as defined in Appendix B.

We expect to see a larger presample size *at worst* not changing the performance of the algorithm in any particular case. There are several instances where the mean number of generations to complete *increased* with a larger presample size. Very few of these represent statistically significant differences; the only place where this can be considered anything but random variation is in the Hierarchical Royal Road problem with uniform crossover. For that unique case, see Appendix B.

[C.1] Data Format

In the tables of data that follow, the seed probability is varied across the *horizontal axis*, so each *column* is a different *probability*. The presample size is varied across the *vertical axis*, so each *row* is a different *presample size*.

When only a number *x* of the thousand runs completed, that is given below the data value as "[*x* runs]". The standard deviation for a value is given alongside each datum point in parentheses, like so: (σ)

The mean number of generations to completion that is presented is calculated from only those runs that did complete. Therefore, the fewer runs of 1000 that completed, the less comparable the resulting mean value is, and it becomes accordingly more important to look at the *number* of runs that completed. To reflect this, all values derived from experiments with *any* non-completing runs are marked with an asterisk * in the normalised tables, and instances where < 90% of runs completed are given on a background of grey in the normalised tables.

Other colours represent the closeness of a particular normalised mean value to the best normalised mean value found. Values presented against a green background are the "best", i.e. the cases where the SGA finished in the fewest generations *after adding normalisation costs*. Values presented against a red background are analogously the "worst". See table C.1 for the details.

< 90% of runs completed
Number of generations greater than 500% of best mean value
Number of generations within 500% of best mean value
Number of generations within 400% of best mean value
Number of generations within 300% of best mean value
Number of generations within 250% of best mean value
Number of generations within 200% of best mean value
Number of generations within 150% of best mean value
Number of generations within 120% of best mean value
Number of generations within 110% of best mean value
Number of generations within 105% of best mean value

Table C.1: Meaning of the colour of a cell for a value in the normalised data tables.

Assuming that each combination of problem and GA parameters has one optimal combination of presample size and seed probability, we would expect to see a pattern of green rings expanding into red rings. However, the problems of noise, the non-granular size of our colour "boxes", and our simplistic normalisation metric mean that we are unlikely to see such a perfect result.

[C.2] Royal Road (RR)

[C.2.1] RR, 2-point Crossover

The CGA completed this problem using a mean 256.6 generations. In the experiments, the seeding algorithm was able to reduce this by as much as 233 generations (90%), or by 208 generations (80%) after normalisation.

Consider tables C.2.1.a and C.2.1.b. After normalisation, the optimal values were a presample size of 80000 – 100000, and a seed probability of 0.2. As expected, we observe a "sweet spot" with these approximately optimal values: the green centre in table C.2.1.b. Note that the seeding algorithm still significantly outperformed the CGA with a seed probability as low as 0.1 or as high as 0.3.

In terms of the mean generations the SGA takes to finish, we run into diminishing returns as we start increasing the presample size beyond 150000. There is no statistically significant difference between a presample of 200000 and a presample of 600000 for any of the seed probabilities.

Obviously at some (unbelievably huge) point the presample would have a strong probability of containing the actual solution, but this is unrealistic to consider in practise. Because there are *vast* differences between the probabilities of containing one block, two blocks, three blocks and so on, there is no visible improvement beyond some particular presample size.

Observe that, in the RR problem[21], the chance of a random individual having at least one correct block is:

$$1 - (1 - (0.5^8))^8) \quad \text{or} \quad 0.030836.$$

A presample of size 90000, which we observe in table C.2.1.b is roughly optimal for our purposes, would have approximately 2774 individuals with at least one correct block, and thus the 50 individuals drawn from the best of the presample would *always* have at least one block each. The seed pool would then be almost certain to contain all 8 blocks.

Now, this would still hold true with a much smaller presample: for instance, the expected contents for a presample of just 1700 is 52 individuals containing at least one block. Given that the higher presample size was optimal, and the performance of the SGA with a presample in the low-thousands was rather poor[22], it would appear that much of the benefit of seeding comes from introducing individuals with *multiple complete blocks*.

Further numerical analysis lends support to this claim. For the RR problem, the chance of a random individual having at least *two* correct blocks is:

$$1 - ((1 - (0.5^8))^8) \ * \ 1 - (1 - (0.5^8))^7) \quad \text{or} \quad 0.000833.$$

A typical presample of 90000 would have approximately 75 individuals with at least two correct blocks, and thus all 50 individuals in the seed pool would almost always have *at least two* blocks. In fact, a presample of 60000 would have an expected value of 50 individuals containing at least two blocks. This is quite close to our optimal presample size, but not quite at the centre of interest in table C.2.1.b.

21 Recall this is the form of Royal Road consisting of 8x8 blocks.
22 But still considerably better than the CGA.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	362.9 (238.1) [857 runs]	314.9 (218.3) [898 runs]	322.6 (222.8) [929 runs]	341.4 (231.8) [874 runs]	505.3 (260.1) [232 runs]
1000	220.1 (192.9) [975 runs]	180.2 (160.8) [989 runs]	168.1 (146.9) [989 runs]	195.1 (164.9) [986 runs]	393.2 (247.3) [733 runs]
2000	123.1 (123.5)	96.0 (80.0) [998 runs]	88.4 (74.5)	104.0 (82.6) [998 runs]	232.8 (185.6) [977 runs]
5000	115.1 (110.4) [997 runs]	90.7 (87.1) [997 runs]	87.3 (70.5)	88.6 (61.8) [999 runs]	222.1 (179.1) [986 runs]
10000	107.7 (98.8) [999 runs]	79.3 (72.8)	72.6 (59.3)	85.8 (64.3) [999 runs]	183.7 (152.6) [995 runs]
20000	87.3 (82.6) [998 runs]	68.6 (60.5)	66.6 (59.3)	72.5 (57.5)	158.3 (136.9) [992 runs]
30000	77.7 (74.6)	59.7 (69.4)	60.5 (57.8)	64.5 (61.5)	130.4 (112.2) [999 runs]
40000	66.1 (63.2)	50.0 (46.7) [999 runs]	49.3 (42.5)	56.0 (47.2)	103.0 (91.2)
50000	57.4 (52.0)	44.3 (39.8)	41.8 (33.3)	47.3 (36.8)	89.1 (74.9)
60000	53.7 (53.3)	41.1 (35.7)	38.8 (38.8)	44.2 (36.8)	77.7 (67.3)
70000	44.0 (44.6)	36.9 (32.2)	35.5 (27.6)	40.7 (38.7)	66.9 (58.1)
80000	42.1 (38.2)	33.6 (30.2)	32.3 (24.6)	36.7 (24.4)	58.9 (46.3)
90000	37.7 (34.6)	32.8 (35.5)	30.8 (23.2)	33.2 (22.0)	55.3 (41.7)
100000	33.3 (28.5)	29.5 (23.2)	26.6 (19.1)	31.9 (20.3)	50.0 (37.0)
125000	29.9 (29.4)	25.1 (18.6)	25.5 (18.3)	27.6 (15.9)	43.3 (40.0)
150000	30.8 (26.6)	24.5 (19.4)	24.0 (15.8)	26.5 (14.7)	41.2 (26.3)
200000	29.8 (27.1)	23.7 (16.1)	23.6 (15.2)	26.2 (14.9)	41.5 (27.6)
300000	28.9 (24.1)	24.2 (17.4)	24.4 (17.0)	27.6 (16.1)	41.1 (26.3)
400000	29.8 (26.8)	23.4 (16.7)	23.4 (14.4)	26.9 (15.9)	38.5 (22.9)
500000	28.7 (25.9)	24.2 (18.8)	24.0 (15.3)	26.6 (16.9)	40.4 (26.1)
600000	27.7 (24.9)	23.8 (18.5)	23.4 (16.2)	26.0 (15.8)	40.4 (25.3)

Table C.2.1.a: Mean number of generations required for the SGA to solve the RR problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	Seed probability				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	363.0 *	315.0 *	322.7 *	341.5 *	505.4 *
1000	220.3 *	180.4 *	168.3 *	195.3 *	393.4 *
2000	123.5	96.4 *	88.8	104.4 *	233.2 *
5000	116.1 *	91.7 *	88.3	89.6 *	223.1 *
10000	109.7 *	81.3	74.6	87.8 *	185.7 *
20000	91.3 *	72.6	70.6	76.5	162.3 *
30000	83.7	65.7	66.5	70.5	136.4 *
40000	74.1	58.0 *	57.3	64.0	111.0
50000	67.4	54.3	51.8	57.3	99.0
60000	65.7	53.1	50.8	56.2	89.7
70000	58.0	50.9	49.5	54.7	80.9
80000	58.1	49.6	48.3	52.7	74.9
90000	55.7	50.8	48.8	51.2	73.3
100000	53.3	49.5	46.6	51.2	70.0
125000	54.9	50.1	50.5	52.6	68.3
150000	60.8	54.5	54.0	56.5	71.2
200000	69.8	63.7	63.6	66.2	81.5
300000	88.9	84.2	84.4	87.6	101.1
400000	109.8	103.4	103.4	106.9	118.5
500000	128.7	124.2	124.0	126.6	140.4
600000	147.7	143.8	143.4	146.0	160.4

Table C.2.1.b: *Normalised* mean number of generations required for the SGA to solve the RR problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

The chance of *three or more* correct blocks in a random individual for the RR problem is:

$$1 - ((1 - (0.5^8))^8) \ * \ 1 - ((1 - (0.5^8))^7) \ * \ 1 - ((1 - (0.5^8))^6) \quad \text{or} \quad 1.93358 \times 10^{-5}.$$

Thus a presample of 90000 would have an average of 1.7 individuals with *at least three* correct blocks. So it would be usual for at least one individual in the seed pool to have at least three blocks, for what we have determined to be the optimal presample size.

[C.2.2] RR, Uniform Crossover

The CGA completed this version of the problem using a mean 246.8 generations (slightly better than the 2-point crossover variant). As can be seen in table C.2.2.a, this achievement was not even matched by the seeding algorithm, let alone before normalisation.

Why did the SGA perform poorly? In both versions of the genetic algorithm, *retention* of blocks will have been poor. This is because the uniform crossover operator affects 64 positions in the bit-string, rather than the two positions of 2-point crossover. Therefore all blocks are likely to be disrupted. But the SGA has no bit-mutation, and can therefore only perform *discovery* through uniform crossover.[23] The probability of discovering a block via uniform crossover is analogous to doing so through eight random mutations, as is the probability of *retaining* a new block being introduced by a seed individual!

Seeding with uniform crossover is acting as a macro-mutation operator. The many possible single-bit micro-mutations of the CGA are apparently better at discovery than this macro-mutation.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	684.8 (208.7) [137 runs]	606.1 (233.5) [466 runs]	- [0 runs]	- [0 runs]	- [0 runs]
5000	607.6 (225.6) [386 runs]	484.8 (236.5) [802 runs]	548.0 (271.3) [10 runs]	- [0 runs]	- [0 runs]
50000	535.6 (259.0) [572 runs]	384.2 (228.6) [933 runs]	501.2 (287.2) [66 runs]	- [0 runs]	- [0 runs]
100000	463.8 (260.8) [744 runs]	294.5 (295.2) [976 runs]	521.3 (252.2) [163 runs]	- [0 runs]	- [0 runs]
200000	445.7 (264.5) [786 runs]	266.9 (185.3) [985 runs]	524.3 (273.4) [208 runs]	- [0 runs]	- [0 runs]
300000	447.2 (258.7) [816 runs]	270.9 (185.3) [983 runs]	505.8 (249.6) [243 runs]	- [0 runs]	- [0 runs]
400000	451.2 (266.1) [797 runs]	284.8 (193.1) [992 runs]	471.3 (261.3) [247 runs]	- [0 runs]	- [0 runs]
500000	440.0 (261.2) [808 runs]	268.3 (185.9) [981 runs]	487.1 (273.2) [252 runs]	- [0 runs]	- [0 runs]
600000	437.7 (260.0) [796 runs]	265.0 (185.6) [990 runs]	519.8 (267.9) [262 runs]	- [0 runs]	- [0 runs]

Table C.2.2.a: Mean number of generations required for the SGA to solve the RR problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

23 When new seed blocks are introduced, they are subject to this same form of crossover before entering the main population.

After normalisation, the optimal values were a presample size of 100000 – 200000, with a seed probability of 0.15. This is a rather larger presample than is optimal for than the version employing 2-point crossover; a better chance at finding multiple blocks in one seed individual is apparently worth the extra computation cost. We presume this is because the algorithm performs so poorly otherwise. We do not present the normalised data here, as too few runs completed for it to be very meaningful.

Note, however, that even though the algorithm failed to beat the CGA, minimal (0.1) seeding was *not* optimal for the SGA here. It may be that the seed probability of 0.15 achieved the necessary balance of

(a) preventing early convergence,
(b) not introducing too much 'junk' from the seed pool, and
(c) acting as a crude mutation operator.

Variations of this seeding rate by as little as 0.05 drastically reduced the success of the algorithm.[24]

[C.2.3] RR, Uniform Crossover with Mutation

The success of the CGA over the SGA for RR with uniform crossover compelled us to investigate a combined form: using the SGA with a 1/128 bit-mutation probability (half the rate of the CGA). We can then take any improvement over the strict seeding algorithm – and any reduction in the optimal seed probability – to provide support for the hypothesis that the poor performance of the SGA is due to it acting as a macro-mutator, which is less effective than the CGA's micro-mutation.

Presample	**Seed probability**			
	0.05	**0.1**	**0.15**	**0.2**
20000	382.3 (193.7) [975 runs]	382.7 (204.8) [972 runs]	494.5 (256.6) [602 runs]	- [0 runs]
50000	357.4 (200.7) [981 runs]	340.7 (198.2) [986 runs]	442.3 (245.6) [780 runs]	- [0 runs]
100000	314.2 (193.5) [990 runs]	280.5 (178.8) [993 runs]	357.8 (226.2) [916 runs]	568.0 (?) [1 runs]
200000	302.0 (182.2) [990 runs]	269.0 (173.3) [995 runs]	339.4 (217.3) [950 runs]	- [0 runs]
500000	302.3 (189.1) [993 runs]	258.3 (169.5) [994 runs]	333.3 (219.7) [954 runs]	607.5 (214.3) [4 runs]

Table C.2.3.a: Mean number of generations required for the SGA to solve the RR problem with uniform crossover and a seed pool of 50, with the addition of mutation. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

24 The difference in table C.2.2.a between the 0.1 and 0.15 seed probability columns, and the 0.15 and 0.2 seed probability columns, is statistically significant for *all* values except the first two rows for the seed probability 0.2. Most values had P values less than 0.0001.

In fact, it became immediately apparent during experimentation that the algorithm was performing even more poorly[25] than the strict SGA with higher seed probabilities. Switching to lower probabilities – that is, making the algorithm more like the CGA – seemed to help.

Recall that the CGA completed this version of the problem using a mean 246.8 generations (slightly better than the 2-point crossover variant). However, according to table C.2.3.a below, the combined algorithm only just approached this level of performance, and that is *before* normalising the data for the cost of the presample. It would appear that, in performing better with a lower seed probability, the algorithm is becoming more "CGA-like", and bit-mutation is driving its ability to find a solution.

Presample	**Seed probability**			
	0.05	**0.1**	**0.15**	**0.2**
20000	386.3 *	386.7 *	498.5 *	-
50000	367.4 *	350.7 *	452.3 *	-
100000	334.2 *	300.5 *	377.8 *	588.0 *
200000	342.0 *	309.0 *	379.4 *	-
500000	402.3 *	358.3 *	433.3 *	707.5 *

Table C.2.3.b: *Normalised* mean number of generations required for the SGA to solve the RR problem with uniform crossover and a seed pool of 50, with the addition of the mutation operator. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[C.3] Staggered Royal Road (R2)

We move on to experimenting on the staggered 8x8 Royal Road problem, R2.

[C.3.1] R2, 2-point Crossover

Using 2-point crossover, the CGA completed this problem using a mean 277.29 generations. The seeding algorithm, conversely, was barely able to solve the problem at all. The rate at which the SGA solved this problem was so low that we shall only consider the number of runs completing as a metric of success. See table C.3.1.a for the raw data.

Due to the staggered nature of the problem, introducing seed individuals along with a 2-point crossover operator is likely to be destructive overall; hence the fact that very few runs actually hit upon the solution. Why was the CGA, for the same parameters, able to solve it in every run?

25 And was almost completely unable to solve the problem

The answer may lie again in the presence of bit-mutation in the CGA. Consider what is happening during any reproduction event. The CGA selects two parent individuals from the main population; the SGA does the same but with a chance of replacing one parent with an individual from the seed pool. The offspring are created by 2-point crossover. Any given block present in parent A will only occur in the offspring if either

(a) the block is present in the parent B, suggesting the block is already commonplace in the main population, or

(b) parent B happens to have the corresponding part of that block – an average of 4 bits – correct. If it does have those bits correct, it is through pure random chance: selection doesn't take into account partial building blocks. Therefore the discovery rates through crossover *or through seeding* are going to be extremely low.

Conversely, in the SGA, consider the case where 2-point crossover brings together several sections of a genome so as to almost form a block, except for a single bit at some arbitrary position within that block. The CGA has a 1/64, or 0.01563, chance of successfully completing the block through mutation. Although it does then have a $(1 – (63/64)^7)$, or 0.10438, chance of destroying the new block through mutation, the chance of 2-point crossover bringing together 7/8 of any block, for a string of 8 staggered blocks, is actually relatively *high* compared to the potential of the SGA for discovery.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	1	14	28	2	0
1000	11	30	67	1	0
2000	19	50	107	12	0
5000	18	70	103	12	0
10000	19	51	131	20	0
20000	26	73	173	21	0
30000	30	94	175	18	0
40000	46	116	216	26	0
125000	112	252	424	76	0
500000	131	278	438	95	0

Table C.3.1.a: Number of runs (of 1000) in which the SGA solved the R2 problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes.

In sum, then, everything the SGA might find useful for this particular combination of problem and parameters is also available to the CGA, but the CGA has the additional advantage in micro-mutation.

[C.3.2] R2, Uniform Crossover

We predicted that substituting in uniform crossover for the SGA on the R2 problem should result in better performance. Uniform crossover, although "jumbling" the genetic information, is still much more capable of successfully crossing two staggered blocks.

Consider parent A, with a block complete, and parent B, without it. Each bit position of the staggered block in which B has a 0 *drastically* reduces the chance of preserving that block in 2-point crossover, but only adds the constraint that the crossover operator choose parent A for uniform crossover. That is, it "only" halves the probability of success each time with uniform crossover, but for any individual B without the block, the bit positions of the block are going to be near-random, and so 2-point crossover will very likely have to place its crossover points such that the vast majority of the genetic material comes from parent A. And, most importantly, if it does this, then it becomes *nearly impossible to pass blocks from parent B on to the offspring*.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	697.9 (215.0) [144 runs]	608.4 (224.2) [445 runs]	- [0 runs]	- [0 runs]	- [0 runs]
1000	662.7 (228.4) [241 runs]	550.7 (228.1) [613 runs]	326.0 (?) [1 run]	- [0 runs]	- [0 runs]
2000	619.2 (242.3) [351 runs]	494.9 (245.8) [793 runs]	442.0 (309.6) [8 runs]	- [0 runs]	- [0 runs]
5000	598.2 (245.4) [401 runs]	486.5 (240.0) [810 runs]	588.6 (326.0) [11 runs]	- [0 runs]	- [0 runs]
10000	583.1 (248.9) [409 runs]	493.9 (239.4) [817 runs]	433.6 (240.4) [11 runs]	- [0 runs]	- [0 runs]
50000	546.2 (256.7) [584 runs]	384.0 (220.0) [917 runs]	474.2 (288.7) [40 runs]	- [0 runs]	- [0 runs]
100000	465.8 (255.9) [739 runs]	308.8 (203.8) [984 runs]	514.5 (281.1) [166 runs]	- [0 runs]	- [0 runs]
200000	458.2 (264.8) [808 runs]	275.0 (193.7) [987 runs]	511.7 (272.2) [223 runs]	- [0 runs]	- [0 runs]
500000	433.7 (263.9) [824 runs]	276.2 (190.5) [985 runs]	487.0 (280.4) [251 runs]	- [0 runs]	- [0 runs]

Table C.3.2.a: Mean number of generations required for the SGA to solve the R2 problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

The CGA completed this problem using a mean 237.1 generations (a little better than using 2-point crossover). While the SGA performed better than it did on the 2-point crossover version in the previous section, it still fails to match the performance of the CGA, even before normalisation. After normalisation, its best performance in terms of lowest adjusted mean-generations-to-completion is 315.0, at a presample of 200000 and seed probability 0.15 – essentially the same as for the RR problem using uniform crossover. Of 1000 runs, 987 solved the problem with these parameters. See table C.3.2.a for details.

[C.4] Hierarchical Royal Road (HRR)

[C.4.1] HRR, 2-point Crossover

The CGA completed this problem with 2-point crossover in a mean 268.3 generations. In the experiments, the seeding algorithm was able to reduce this by as much as 233 generations (94%), or by 220 generations (80%) after normalisation. This is an excellent result comparable to the non-hierarchical version.

From a normalised perspective (see table C.4.1.b), the optimal values were a presample size of 80000 – 100000, with a seed probability of 0.2 or slightly lower. As with the RR problem, the seeding algorithm still outperformed the CGA with seed probabilities that deviated from the optimal.

For the RR problem, the SGA did not improve to a statistically significant level when increasing the presample size from 200000 to 500000. By contrast, the same increase for the HRR problem improved the results by between 5.2 and 15.5 generations – a statistically significant increase[26] for each seed probability tested.

The pertinent difference here is that the hierarchical problem rewards individuals with *adjacent* blocks more. Increasing the presample size becomes nearly useless for the non-hierarchical version because the probability of ever getting more than three blocks in a random individual is negligible at 'reasonable' scales. For the hierarchical version, even though increasing the presample size does not improve the potential fitness by increasing the probability of getting large numbers of blocks in one individual, it *does* increase the probability of getting more *contiguous* blocks in one individual, and so is visibly useful for longer.

26 Highly statistically significant, in fact, with P values < 0.0001.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	357.9 (245.0) [855 runs]	298.3 (227.6) [931 runs]	302.5 (217.8) [940 runs]	392.5 (258.9) [712 runs]	512.2 (271.8) [63 runs]
1000	189.2 (173.3) [976 runs]	150.9 (135.7) [989 runs]	148.8 (129.3) [996 runs]	234.4 (195.9) [964 runs]	454.7 (282.3) [360 runs]
2000	112.1 (110.5) [999 runs]	88.8 (76.4) [999 runs]	84.9 (78.6)	122.3 (103.7)	358.3 (253.8) [814 runs]
5000	105.6 (98.8) [999 runs]	83.3 (74.2) [999 runs]	77.0 (60.3) [999 runs]	113.3 (90.1)	333.5 (244.1) [836 runs]
10000	104.2 (112.6)	77.5 (65.1)	75.9 (63.6)	100.0 (89.0) [999 runs]	322.6 (240.0) [874 runs]
20000	86.3 (95.2)	65.8 (60.6) [999 runs]	61.9 (47.5)	86.3 (66.1)	268.4 (219.7) [941 runs]
30000	73.0 (71.4) [999 runs]	58.8 (52.7) [999 runs]	58.1 (49.7)	75.2 (57.7)	222.7 (198.8) [967 runs]
40000	63.7 (62.9)	54.0 (53.5)	50.9 (37.8)	65.7 (49.1)	189.3 (176.4) [992 runs]
50000	51.6 (50.3)	47.3 (44.2)	45.5 (38.6)	57.0 (40.2)	149.9 (146.2) [995 runs]
60000	50.1 (47.5)	40.3 (31.3)	41.3 (31.2)	52.1 (38.9)	129.1 (123.5) [996 runs]
70000	45.0 (41.0)	36.9 (27.7)	37.5 (24.1)	47.4 (35.7)	109.9 (113.6) [997 runs]
80000	39.9 (37.0)	33.9 (28.6)	33.7 (25.7)	44.3 (32.2)	91.4 (83.8)
90000	38.2 (35.5)	32.2 (26.1)	30.4 (20.6)	39.1 (27.7)	82.3 (82.9) [999 runs]
100000	33.6 (27.4)	31.2 (24.4)	29.0 (19.2)	35.7 (23.4)	70.9 (59.5)
125000	29.9 (25.3)	25.6 (18.2)	27.0 (17.5)	31.8 (19.2)	57.5 (51.5)
150000	28.7 (24.0)	25.9 (22.0)	24.5 (15.1)	29.9 (18.3)	53.0 (47.1)
200000	26.5 (32.2)	22.7 (15.4)	24.1 (16.0)	28.6 (18.0)	47.9 (36.6)
500000	19.1 (12.8)	17.5 (10.5)	18.8 (11.7)	21.1 (12.2)	32.4 (27.2)

Table C.4.1.a: Mean number of generations required for the SGA to solve the HRR problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	Seed probability				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	358.0 *	298.4 *	302.6 *	392.6 *	512.3 *
1000	189.4 *	151.1 *	149.0 *	234.6 *	454.9 *
2000	112.5 *	89.2 *	85.3	122.7	358.7 *
5000	106.6 *	84.3 *	78.0 *	114.3	334.5 *
10000	106.2	79.5	77.9	102.0 *	3246 *
20000	90.3	69.8 *	65.9	90.3	272.4 *
30000	73.6 *	59.4 *	64.1	81.2	223.3 *
40000	71.7	62.0	58.9	73.7	197.3 *
50000	61.6	57.3	55.5	67.0	159.9 *
60000	62.1	52.3	53.3	64.1	141.1 *
70000	59.0	50.9	51.5	61.4	123.9 *
80000	55.9	49.9	49.7	60.3	107.4
90000	56.2	50.2	48.4	57.1	100.3 *
100000	53.6	51.2	49.0	55.7	90.9
125000	54.9	50.6	52.0	56.8	82.5
150000	58.7	55.9	54.5	59.9	83.0
200000	66.5	62.7	64.1	68.6	87.9
500000	119.1	117.5	118.8	121.1	132.4

Table C.4.1.b: *Normalised* mean number of generations required for the SGA to solve the HRR problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[C.4.2] HRR, Uniform Crossover

The CGA again performed slightly better on the HRR problem with uniform crossover, completing in an average of 230.5 generations. The SGA did not come close to this result. This follows the trend of the RR and R2 problems, where uniform crossover worked slightly better for the CGA and rather poorly for the seeding algorithm.

Some of these results were somewhat incongruous. See Appendix B for a discussion of them.

[C.5] One-Sum (OS)

We turn to our baseline "sanity check" problem, One-Sum. We do not expect any difficulties with it. Recall that the fitness of an individual in the OS problem is simply *m*, where *m* is the number of its 64 bits set to 1.

[C.5.1] OS, 2-point Crossover

The CGA was able to complete this simple problem in a mean 18.2 generations with 2-point crossover. In our experiments, the seeding algorithm was only able to beat this performance with the very highest presample sizes, and did not even match the result after normalisation.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	45.4 (23.8)	43.0 (19.7)	45.4 (19.1)	63.7 (25.9)	245.3 (181.7) [983 runs]
1000	39.0 (19.8)	37.2 (17.2)	40.6 (16.3)	53.0 (19.5)	178.5 (141.9) [997 runs]
2000	34.5 (18.7)	32.6 (13.9)	36.2 (14.2)	48.1 (17.9)	130.1 (92.5) [999 runs]
3000	32.2 (15.5)	30.9 (13.8)	33.7 (12.7)	43.7 (16.6)	113.4 (78.4)
4000	31.0 (15.7)	29.6 (12.4)	32.8 (12.9)	42.9 (16.1)	100.8 (67.1)
5000	30.2 (14.7)	27.9 (12.0)	30.9 (11.6)	40.8 (14.7)	92.0 (58.8)
6000	28.8 (14.6)	27.5 (12.2)	30.7 (11.3)	39.7 (14.7)	89.8 (58.0)
8000	28.6 (14.9)	25.8 (10.7)	28.7 (10.2)	38.0 (14.0)	83.2 (53.6)
10000	26.7 (12.9)	26.6 (11.7)	28.5 (10.5)	37.6 (13.6)	78.1 (48.1)
20000	24.1 (12.0)	23.6 (9.5)	25.7 (8.9)	33.1 (11.4)	64.8 (36.7)
30000	22.7 (11.8)	22.4 (9.4)	24.1 (8.7)	30.9 (11.4)	57.3 (31.3)
40000	21.6 (10.6)	21.6 (8.3)	23.7 (8.3)	30.4 (11.3)	55.8 (31.5)
50000	21.1 (10.0)	20.7 (7.8)	22.8 (8.1)	29.1 (10.2)	53.1 (29.8)
100000	19.8 (6.4)	19.0 (5.6)	21.3 (5.7)	26.9 (7.2)	45.5 (15.2)
500000	16.1 (8.5)	16.3 (7.0)	17.8 (7.0)	21.7 (9.2)	34.0 (22.9)

Table C.5.1.a: Mean number of generations required for the SGA to solve the OS problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

From a normalised perspective, as seen in table C.5.1.b below, the optimal values were a presample size of 10000 – 20000, with a seed probability of 0.1 – 0.5. However, the best results were not tightly bound to a particular seed probability or presample size. Consider table C.5.1.a above. We conclude that having a

large enough presample and low enough seed probability to make sure the seeding is not badly *harming* the main population by introducing seed individuals is enough, but seeding *per se* is not very helpful.

Over the course of any reasonably tractable GA problem, seeding should begin as a very useful operation, because the main population is random at the beginning: every single seed individual should be better than a random one. As the run continues, the utility of seeding *may* fall as more blocks are present in the main population – although it will always be useful to prevent early convergence.[27]

But in the OS problem, there are no blocks[28]. Within just a few generations, the main population should have reached the fitness of the seed pool, and at that point, seeding may in fact begin to hinder the process. With a presample as large as 500000, the seed pool will contain individuals of such high fitness that they can be quickly assembled into the final solution, and so (when not normalising for the cost of seed pool construction) the algorithm is able to beat the CGA. Note that a presample of 500000 is expected to generate an average 20.4 individuals with 50 or more bits set to 1.[29] It is a relatively short step from here to the solution.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	45.5	43.1	45.5	63.8	245.4 *
1000	39.2	37.4	40.8	53.2	178.7 *
2000	34.9	33.0	36.6	48.5	130.5 *
3000	32.8	31.5	34.3	44.3	114.0
4000	31.8	30.4	33.6	43.7	101.6
5000	31.2	28.9	31.9	41.8	93.0
6000	30.0	28.7	31.9	40.9	91.0
8000	30.2	27.4	30.3	39.6	84.8
10000	28.7	28.6	30.5	39.6	80.1
20000	28.1	27.6	29.7	37.1	68.8
30000	28.7	28.4	30.1	36.9	63.3
40000	29.6	29.6	31.7	38.4	63.8
50000	31.1	30.7	32.8	39.1	63.1
100000	49.8	49.0	51.3	56.9	75.5
500000	116.1	116.3	117.8	121.7	134.0

Table C.5.1.b: *Normalised* mean number of generations required for the SGA to solve the OS problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

27 In fact, we did *not* observe a decrease in usefulness of seeding in our tests of seeding functions in section 4.1.
28 An argument could be made that there are 64 "blocks" of length 1, but it amounts to the same thing.
29 Or 0.4 individuals with 51 or more bits set to 1, *etcetera*.

[C.5.2] OS, Uniform Crossover

When we use uniform crossover, the CGA improves its performance from a mean 18.2 generations to complete, to 12.2 generations. Usually uniform crossover would be disruptive when working with multi-bit blocks, but these do not exist for this problem. Instead, the crossover operator is able to address 64 of 64 possible crossover points each time, instead of just 3. The advantage is obvious.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	15.1 (7.0)	17.6 (5.2)	30.1 (13.3)	245.3 (207.2) [979 runs]	513.7 (274.1) [27 runs]
1000	14.1 (5.7)	16.5 (4.3)	26.2 (10.1)	158.9 (134.4) [999 runs]	491.7 (278.7) [77 runs]
2000	13.1 (4.0)	15.2 (3.5)	23.7 (8.4)	107.1 (87.8)	471.6 (279.8) [173 runs]
3000	12.6 (4.4)	14.8 (3.6)	22.3 (8.2)	93.0 (73.4)	447.4 (294.4) [249 runs]
4000	12.7 (3.9)	14.5 (3.4)	21.6 (7.5)	77.2 (60.1)	473.3 (288.1) [352 runs]
5000	12.4 (3.5)	14.3 (3.1)	20.9 (7.3)	75.8 (59.3)	461.5 (280.6) [443 runs]
6000	12.1 (3.2)	14.0 (3.0)	20.7 (7.2)	66.0 (49.8)	447.6 (279.0) [517 runs]
8000	11.9 (3.0)	13.7 (2.8)	19.7 (6.4)	59.8 (43.2)	454.5 (282.6) [567 runs]
10000	11.8 (3.1)	13.5 (2.6)	18.9 (5.7)	56.1 (40.4)	424.1 (286.9) [643 runs]
20000	11.2 (2.2)	12.9 (2.6)	17.6 (5.6)	43.5 (30.4)	355.5 (261.0) [850 runs]
30000	11.1 (2.5)	12.7 (2.5)	16.8 (5.1)	38.7 (23.9)	311.1 (238.8) [925 runs]
50000	10.7 (1.9)	12.1 (2.4)	16.1 (4.3)	35.2 (22.7)	253.8 (220.5) [962 runs]
100000	10.4 (1.8)	11.8 (2.2)	15.2 (4.2)	28.9 (15.6)	181.7 (169.4) [999 runs]
500000	9.7 (1.4)	10.6 (1.8)	13.0 (3.1)	21.0 (10.3)	76.8 (64.7)

Table C.5.2.a: Mean number of generations required for the SGA to solve the OS problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

From table C.5.2.a, we see that the SGA performs similarly to the 2-point version. However, a smaller presample size and lower seeding probability appear to be optimal after normalisation (see table C.5.2.b). Because all the bits needed are present in the main population, uniform crossover should be sufficient to combine them; the only role seeding (or mutation, in the case of the CGA) needs to play is in preventing early convergence.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	15.2	17.7	30.2	245.4 *	513.8 *
1000	14.3	16.7	26.4	159.1 *	491.9 *
2000	13.5	15.6	24.1	107.5	472.0 *
3000	13.2	15.4	22.9	93.6	448.0 *
4000	13.5	15.3	22.4	78.0	474.1 *
5000	13.4	15.3	21.9	76.8	462.5 *
6000	13.3	15.2	21.9	67.2	448.8 *
8000	13.5	15.3	21.3	61.4	456.1 *
10000	13.9	15.5	20.9	58.1	426.1 *
20000	15.2	16.9	21.6	47.5	359.5 *
30000	16.1	18.7	22.8	44.7	317.1 *
50000	20.7	22.1	26.1	45.2	263.8 *
100000	30.4	31.8	35.2	48.9	201.7 *
500000	109.71	110.6	113.0	121.0	176.8

Table C.5.2.b: *Normalised* mean number of generations required for the SGA to solve the OS problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[C.6] Hierarchical If-And-Only-If (HIFF)

The CGA was able to outperform the SGA on OS because it was a simple problem not involving building blocks. Let us now turn to a much more difficult problem with building blocks that are both prominent and massively interdependent. The fitness landscape of HIFF is akin to a fractal, with two equidistant maximum peaks, then for each of those, two peaks half their height on either side, and so on. Recall that the CGA is essentially *unable* to solve this problem.

[C.6.1] HIFF, 2-point Crossover

When using 2-point crossover, one in a thousand runs of the CGA solved the problem. The “weaker” selection operators – rank selection and 2-tournament – performed a little better, but it is obvious that the canonical genetic algorithm is going to perform poorly on a problem essentially *made* of local optima. Rank selection and 2-tournament are more able to escape maxima, because they are much less likely to select the 'best' individual from the whole population (i.e. the one trapped in local maxima).

The results of the seed algorithm (table C.6.1.a) are quite striking. Uniquely of our tests so far, the SGA performs best with the highest seed probabilities. Let us examine this behaviour in detail. The first thing to point out is that we should expect some (possibly very slight) evidence of a growing preference for higher seed probabilities with larger presamples *whenever* seeding is a viable technique, simply because a larger presample means we are stocking our seed pool with fitter individuals, so seeding becomes a better strategy.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	372.7 (277.2) [77 runs]	369.7 (283.9) [106 runs]	350.2 (280.5) [128 runs]	486.4 (272.5) [305 runs]	582.2 (244.6) [56 runs]
1000	302.9 (255.2) [113 runs]	333.2 (260.3) [114 runs]	332.3 (258.3) [153 runs]	462.7 (270.8) [376 runs]	549.0 (265.9) [125 runs]
10000	274.0 (245.6) [194 runs]	208.6 (228.3) [182 runs]	228.8 (224.3) [242 runs]	387.1 (274.1) [506 runs]	474.0 (267.8) [358 runs]
100000	249.2 (268.8) [363 runs]	253.4 (263.2) [408 runs]	234.7 (245.6) [429 runs]	302.6 (265.4) [674 runs]	380.9 (278.7) [697 runs]
200000	257.0 (256.6) [450 runs]	244.6 (265.1) [476 runs]	236.7 (249.5) [541 runs]	273.5 (249.4) [740 runs]	325.5 (260.6) [837 runs]
300000	259.6 (261.4) [517 runs]	212.7 (228.7) [532 runs]	215.3 (236.1) [565 runs]	258.7 (250.2) [783 runs]	295.2 (241.4) [848 runs]
400000	216.4 (245.8) [505 runs]	197.8 (221.0) [586 runs]	216.2 (232.1) [624 runs]	251.1 (262.1) [820 runs]	267.8 (235.5) [916 runs]
500000	212.9 (231.2) [551 runs]	202.0 (213.5) [569 runs]	190.4 (208.7) [637 runs]	234.9 (242.9) [789 runs]	269.8 (236.4) [929 runs]
750000	208.4 (229.9) [550 runs]	196.6 (218.8) [601 runs]	190.7 (220.6) [687 runs]	227.6 (245.3) [820 runs]	233.8 (218.8) [935 runs]

Table C.6.1.a: Mean number of generations required for the SGA to solve the HIFF problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

At the point where our presample is large enough to include *the actual solution* in our seed pool, the optimal seeding probability would of course be 1. And as we have seen, seeding with random individuals

is less effective than the CGA, so at that point, we would want a seeding probability close to zero.

This preference for higher seed probabilities will only hold as long as our fitness measure is a good one. For example, we have seen that the success of the SGA on the HIFF problem depends more on the number of Royal Road -style elementary blocks than it does on the HRR fitness function.
Our conjecture is that seeding is particularly suited to the HIFF problem with *n*-point crossover, because bit-mutation will essentially *never* free it from (sufficiently large) local maxima. Conversely, with seeding, as long as seed individuals are introduced into the main population in sufficiently large numbers, it should be possible to either combine those individuals into partial solutions rivalling the fitness of any partial solution the main population has found, or combine those individuals with the partial solution of the main population to improve it.

Given the complex nature of the problem, experiments running for a greater number of generations, with higher seed probabilities, are indicated. Currently, even considering the rank or 2-tournament selection parameters for the CGA (which give 7-9% completion rate, and which might improve the seeding algorithm even more), we are still easily beating the CGA results.

[C.6.2] HIFF, Uniform Crossover

Again, the CGA is unable to solve this problem-parameter combination within 1000 generations, although this can be brought up to occasional successes (3-4% of runs completing) with 2-tournament or rank selection, the “rugged” selection operators helping escape local maxima.

As can be seen in table C.6.2.a, on the uniform variant, the SGA is just barely effective. It seldom finds the solution and generally performing considerably worse than it does using 2-point crossover. However, it still beats the CGA with these parameters.

It is unclear whether the seeding algorithm simply needs more time to solve this particular problem (due to the destructive nature of uniform crossover), or whether it is usually getting trapped in one of the many local maxima. The fact that we no longer seem to be seeing the preference for greater seed probabilities which cropped up in the previous section (although we cannot state this with statistical rigour) suggests that the seeding is fairly ineffective. Again, longer-running experiments are indicated.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	0	10	1	0	0
5000	2	4	1	0	0
50000	1	15	3	0	0
500000	1	28	3	0	0

Table C.6.2.a: Number of runs (of 1000) in which the SGA solved the HIFF problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes.

[C.7] Deceptive Trap (DT)

We now move on to what we thought would be our hardest problem, as it is the only one featuring deception. DT, our form of Deceptive Trap, has a solution consisting of 12 blocks of length 5, each block corresponding to a trap.

We should point out that the CGA simply *cannot* solve this deceptive problem: it did not have a single successful run, in the thousand runs for each of the fifteen combinations of our five selection operators and three crossover operators, or for any of the intermediate tests that are not discussed in these results.

Any result at all from the SGA could therefore be seen as an improvement. With this in mind, it is remarkable that the seeding algorithm actually does rather well. As we shall see, the SGA performs better on DT than it does on HIFF.

[C.7.1] DT, 2-point Crossover

As table C.7.1.b shows, a seed probability of 0.15 to 0.2, and a relatively large presample of 100000 to 300000, appears optimal after normalisation. Table C.7.1.a demonstrates that the SGA is sufficiently powerful to solve the DT problem 1000 times out of 1000 for a presample of 500000.

For the CGA to discover a block B, it would need to

(a) Have a block B of penultimate fitness (4), and flip all five of its bits at once via mutation.

(b) Or have a block B of fitness 3 or 2 or 1 or 0, and flip 4 or 3 or 2 or 1 of its bits respectively via mutation; such an individual would be increasingly unlikely to survive in the main population as those cases where fewer of its bits need changing to reach maximum fitness correspond to cases where it is assigned lower and lower fitness by the DT operator.

(c) Or discover B by chance crossover of two individuals with 1-bits in portions of B; such individuals would be *less* likely to survive and *less* likely to be selected for crossover due to possessing these 1-bits.

(d) Or mutate from the original population when a cluster of 1-bits randomly occur in a block, having not yet been subject to selection, and only a few more need to be set to 1.

So how does the SGA solve this otherwise GA-intractable problem? It only needs to introduce an individual with that block already solved, and 2-point crossover will do the rest. Even if there are other seed individuals which have blocks C...N solved and block B in the penultimate state, and are therefore fitter than the individual with block B solved, then each seeding event has a 1/50 chance of reintroducing the block B, so at *some* point, 2-point crossover is going to successfully graft it into the genome in the main population such that it propagates itself.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	598.1 (214.6) [451 runs]	582.2 (216.6) [486 runs]	609.3 (224.5) [461 runs]	615.9 (219.6) [75 runs]	496.3 (458.0) [3 runs]
1000	552.0 (232.4) [648 runs]	533.2 (230.2) [682 runs]	576.1 (226.3) [638 runs]	585.7 (241.4) [151 runs]	425.5 (229.2) [6 runs]
10000	370.6 (213.9) [930 runs]	323.8 (190.6) [960 runs]	355.3 (200.50 [947 runs]	488.4 (244.8) [612 runs]	478.4 (274.4) [37 runs]
50000	265.1 (182.1) [984 runs]	223.2 (151.2) [989 runs]	253.8 (165.6) [989 runs]	377.9 (241.4) [875 runs]	503.7 (271.8) [171 runs]
100000	216.4 (150.4) [993 runs]	197.5 (140.0) [997 runs]	201.4 (129.0) [994 runs]	318.6 (218.7) [932 runs]	470.1 (260.0) [240 runs]
200000	183.2 (131.7) [994 runs]	168.5 (126.0) [999 runs]	168.4 (112.2) [997 runs]	276.7 (205.9) [962 runs]	481.7 (278.4) [346 runs]
300000	174.0 (126.6) [999 runs]	143.7 (95.4) [999 runs]	161.3 (107.4)	244.9 (185.7) [986 runs]	436.0 (282.3) [423 runs]
400000	160.0 (119.4) [999 runs]	135.7 (101.4)	148.0 (98.9)	231.6 (183.7) [992 runs]	427.6 (272.0) [484 runs]
500000	153.5 (111.6) [999 runs]	128.3 (85.1)	138.8 (101.5)	228.3 (174.7) [989 runs]	430.9 (279.6) [541 runs]

Table C.7.1.a: Mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	598.2 *	582.3 *	609.4 *	616.0 *	496.4 *
1000	552.2 *	533.4 *	576.3 *	585.9 *	425.7 *
10000	372.6 *	325.8 *	357.3 *	490.4 *	480.4 *
50000	275.1 *	233.2 *	263.8 *	387.9 *	513.7 *
100000	236.4 *	217.5 *	221.4 *	338.6 *	490.1 *
200000	223.2 *	208.5 *	208.4 *	316.7 *	521.7 *
300000	234.0 *	203.7 *	281.3	304.9 *	496. *
400000	240.0 *	295.7	308.0	311.6 *	507.6 *
500000	253.5 *	328.3	338.8	328.3 *	530.9 *

Table C.7.1.b: *Normalised* mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[C.7.2] DT, Uniform Crossover

Interestingly, just like the CGA, the SGA cannot solve the DT problem with uniform crossover. Not a single run completed in our experiments.

The intuitive explanation for this is that uniform crossover is completely unsuited to handling deception. We illustrate this with an example. Consider the case where an individual X with block B completed (fitness 5) is crossed with an individual Y with block B in the state of penultimate fitness (maximum deception; fitness 4). The *only* way that the offspring will retain block B is if it inherits all its bits for that block from parent X. This has a $1/2^5$, or 0.031, chance of happening. We expect, on average, half the bits to come from X and half to come from Y, giving us either a string of three 1-bits and two 0-bits (fitness 1), or two 1-bits and three 0-bits (fitness 2).

The prospects are slightly better if Y has block B in sub-penultimate fitness (a single 1-bit), but this is correspondingly less likely to happen, since Y would be correspondingly less fit and less likely to (a) survive in the population and (b) be selected.

Given that the likelihood of destroying a block far outstrips the likelihood of propagating one, let alone discovering one in the first place, it is unsurprising that the SGA fails on this problem.

[C.8] Long Tests of Difficult Problems

The SGA had some interesting behaviour on the DT and HIFF problems. Because of this, we repeated some of our experiments, running each one for up to 20000 generations (a twentyfold increase). Due to time constraints, however, we were forced to repeat each experiment over only 100 runs, not 1000. This is enough to still give us a large measure of statistical confidence in our results.

[C.8.1] Long HIFF

As a baseline, the CGA was run on HIFF with 2-point crossover for 20000 generations. None of the 100 runs completed. This experiment was repeated a total of six times; one run completed on generation 18 and one on generation 7711. We imagine that the former stumbled upon the global maximum by chance in the early stages of the algorithm. The latter is slightly more inexplicable, but we suggest that a very close approximation of the solution was found, and chance mutation drove it to completion later on.

The SGA performed best on HIFF with higher seed probabilities, so we used probabilities 0.2, 0.25, 0.3, and 0.35 for our long experiments. Judging by table C.8.1.b, a seed probability of 0.3 and presample near 500000 are our optimal values after normalisation. The seeding probability appears to be rather sensitive in this problem. It is interesting that the optimal value for it is close to what would ensure a half-chance that at least one parent in a given reproduction event would be replaced by a seed individual.

Practically every combination of parameters tested worked better than the CGA on this problem, as can be seen in table C.8.1.a. For presamples of at least 10000, we can say with confidence that the SGA outperformed the CGA even with its least suitable seeding probabilities.

Presample	**Seed probability**			
	0.2	**0.25**	**0.3**	**0.35**
100	8016.2 (6701.3) [20 runs]	3932.3 (3836.3) [97 runs]	7794.4 (5750.6) [25 runs]	- [0 runs]
500	5028.9 (5578.9) [42 runs]	2860.1 (3457.0)	6793.6 (5497.9) [69 runs]	6652.0 (1402.3) [6 runs]
1000	6519.9 (6824.4) [42 runs]	2147.8 (2469.9)	6299.6 (4726.2) [77 runs]	10150.4 (6127.7) [8 runs]
5000	5773.6 (6549.1) [46 runs]	2253.9 (2827.5) [99 runs]	4311.0 (4142.7) [96 runs]	8792.3 (4801.8) [29 runs]
10000	4223.6 (4998.5) [37 runs]	1758.4 (2295.0)	3072.2 (3979.8) [98 runs]	9394.3 (6405.1) [44 runs]
25000	3149.6 (4507.9) [60 runs]	1618.7 (2168.7)	2039.7 (2762.4)	7191.3 (5398.5) [56 runs]
50000	3231.2 (4810.0) [76 runs]	1212.9 (1838.9)	1941.3 (3233.8) [98 runs]	5993.0 (5184.1) [80 runs]
100000	3349.0 (4952.66) [72 runs]	1324.6 (2586.6)	953.1 (1450.1)	4247.9 (4834.4) [91 runs]
200000	1421.6 (3682.1) [76 runs]	813.9 (1216.0)	945.2 (2363.7)	2669.1 (3273.9) [98 runs]
300000	2376.9 (4976.9) [84 runs]	998.9 (1969.5)	615.7 (966.9)	1896.5 (2549.2) [97 runs]
400000	2772.8 (4665.0) [78 runs]	769.7 (1179.9)	474.4 (740.4)	1754.8 (2890.1) [99 runs]
500000	1724.3 (4218.4) [81 runs]	555.6 (910.2)	320.6 (443.3)	1451.2 (2418.1)
600000	2043.2 (4443.0) [86 runs]	597.1 (849.5) [99 runs]	368.6 (382.2)	1136.9 (1145.1)
750000	1756.2 (3793.7) [86 runs]	629.7 (913.9)	308.0 (369.4)	1239.0 (1980.6)

Table C.8.1.a: Mean number of generations required for the SGA to solve the HIFF problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 100 runs completed within 20000 generations. Standard deviations are given in parentheses (σ).

Presample	Seed probability			
	0.2	**0.25**	**0.3**	**0.35**
100	8016.2 *	3932.3 *	7794.5 *	-
500	5029.0 *	2861.1	6793.7 *	6652.1 *
1000	6521.1 *	2148.0	6299.8 *	10150.6 *
5000	5774.6 *	2254.9 *	4312.0 *	8793.3 *
10000	4225.6 *	1760.4	3074.2 *	9396.3 *
25000	3154.6 *	1623.7	2044.7	7196.3 *
50000	3241.2 *	1222.9	1951.3 *	6003.0 *
100000	3369.0 *	1344.6	973.13	4267.9 *
200000	1461.6 *	853.9	985.2	2709.1 *
300000	2436.9 *	1058.9	675.7	1956.5 *
400000	2852.8 *	849.7	554.4	1834.8 *
500000	1824.3 *	655.62	420.6	1551.2
600000	2163.2 *	617.1 *	488.6	1256.9
750000	1906.2 *	779.7	458.0	1389.0

Table C.8.1.b: *Normalised* mean number of generations required for the SGA to solve the HIFF problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 100 runs completed within 20000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

We did attempt to use uniform crossover as well. As expected, none of the runs of the CGA with uniform crossover solved the HIFF problem within 20000 generations. Of the 800 total runs across eight experiments we performed with the SGA and uniform crossover, three solved the problem, presumably by a statistical fluke. The generations at which the solution was found were rather late: gen 14636, gen 17454 and gen 4645.

[C.8.2] Long DT

We ran the CGA with 2-point crossover for 20000 generations, and it failed to solve the DT problem in any of 100 runs. This experiment was repeated five more times and it failed each time. We then ran the SGA, testing with the lower seed probabilities it seemed to work with. By the time we reached a presample of just 25000, the SGA was finding the solution 100% of the time, with all seed probabilities. As table C.8.2.b shows, there was no specific combination of seed probability and presample size that worked particularly well; instead the SGA had a generally good performance with presamples from 100000 to 500000 and 0.1 to 0.2 seed probability.

Indeed, the results are quite promising. They appear to have drastically larger running times those in table C.7.1.a, but this is simply a reflection of the fact that the averages here are taken over something closer to a *genuine* sample of the population, whereas in the shorter experiment they were only taken over those runs that *did* finish within 1000 generations.

Presample	**Seed probability**		
	0.1	**0.15**	**0.2**
100	3182.6 (3481.8) [60 runs]	4279.1 (4788.5) [65 runs]	4683.9 (4269.9) [76 runs]
500	1458.1 (2151.1) [99 runs]	1622.2 (1968.1) [94 runs]	1565.3 (2092.3) [95 runs]
1000	1061.4 (1007.8) [98 runs]	1249.5 (2150.9) [96 runs]	965.1 (1154.4) [99 runs]
5000	577.2 (538.3)	555.2 (505.4) [99 runs]	548.4 (632.3)
10000	498.0 (467.9)	370.7 (248.7) [98 runs]	442.4 (335.9)
25000	362.8 (606.2)	355.7 (317.3)	292.4 (206.7)
50000	293.9 (267.3)	204.7 (125.6)	229.3 (150.3)
75000	238.4 (171.5)	215.7 (292.8)	208.3 (130.6)
100000	214.7 (174.4)	203.8 (302.4)	200.2 (122.4)
125000	213.9 (170.8)	181.6 (118.1)	205.8 (135.3)
150000	257.9 (435.0)	163.2 (136.9)	183.1 (117.5)
175000	180.9 (122.0)	184.0 (151.1)	162.1 (100.9)
200000	178.9 (112.6)	167.0 (106.1)	168.6 (105.3)
300000	163.9 (118.2)	147.4 (108.2)	168.4 (117.5)
400000	172.7 (177.4)	133.9 (96.5)	146.4 (94.2)
500000	149.0 (105.4)	132.3 (86.5)	158.0 (114.1)
600000	161.7 (122.3)	121.1 (80.1)	141.2 (87.7)

Table C.8.2.a: Mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 100 runs completed within 20000 generations. Standard deviations are given in parentheses (σ).

Remarkably, even when a seed pool of 50 was drawn from a presample of 100, two-thirds of the runs were able to solve the DT problem within 20000 generations. This is particularly amazing, because such a seed pool would be filled with individuals containing blocks in various 'deceptive' states. Consider that the chance of a random individual having at least one *correct* block is:

$$1 - (1 - (0.5^5))^{12}) \quad \text{or} \quad 0.31681$$

So a presample of size 100 would have an average of 32 individuals with one or more blocks correct. However, there is no guarantee that these would be selected for the test pool: individuals with no blocks completely correct, but multiple blocks in high-fitness (and therefore highly deceptive) states would likely outweigh them.

Presample	**Seed probability**		
	0.1	**0.15**	**0.2**
100	3182.6 *	4279.1 *	4683.9 *
500	1458.2 *	1622.3 *	1565.4 *
1000	1061.6 *	1249.7 *	965.3 *
5000	578.2	556.2*	549.5
10000	500.0	372.7 *	444.4
25000	362.8	355.7	292.4
50000	303.9	214.7	239.3
75000	253.4	230.7	223.3
100000	234.7	223.8	220.2
125000	238.9	206.6	230.8
150000	287.9	193.2	213.1
175000	215.9	219.0	197.1
200000	218.9	207.0	208.6
300000	223.9	207.4	228.4
400000	252.7	213.9	226.4
500000	249.0	232.3	258.0
600000	281.7	241.1	261.22

Table C.8.2.b: *Normalised* mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. An asterisk * indicates that not all of the 100 runs completed within 20000 generations; entries where fewer than 90% of the runs completed are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

Neither the CGA nor the SGA had any successes whatsoever in running on the DT problem with uniform crossover. Again, uniform crossover is completely unsuited to handling the maximally deceptive nature of a block.

Appendix D: Determining Optimal Presample Size and Seed Pool Size

We present the results in the same data formats we did for previous experiments. See section C.1 for clarification.

[D.1] Royal Road (RR)

[D.1.1] RR, 2-point Crossover

Using a seed probability of 0.2, we ran experiments on our default RR problem. Our initial seed pool choice of size 50, taken from [Skinner, 2009, pp 92-93], appears to have been a good choice. 50 to 100 seems to be optimal (see table D.1.1.b). We can see from this and table D.1.1.a that 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. This improvement was statistically significant for each presample size tested.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	495.1 (269.8) [131 runs]	133.2 (134.0) [983 runs]	168.1 (146.9) [989 runs]	245.7 (174.0) [987 runs]	421.1 (217.2) [939 runs]	N/A
5000	395.7 (292.3) [197 runs]	127.1 (140.7) [984 runs]	87.3 (70.5)	70.2 (51.0)	179.6 (112.0) [998 runs]	277.8 (165.5) [998 runs]
10000	343.7 (296.1) [258 runs]	109.1 (128.3) [991 runs]	72.6 (59.3)	68.3 (49.2)	97.5 (62.4)	175.8 (110.4)
50000	251.8 (244.6) [388 runs]	40.3 (51.2)	41.8 (33.3)	48.0 (34.7)	58.4 (35.0)	61.1 (36.3)
100000	272.5 (269.5) [396 runs]	33.5 (37.4) [999 runs]	26.6 (19.1)	35.6 (21.7)	55.3 (36.1)	58.3 (35.1)
500000	193.9 (241.5) [472 runs]	29.0 (23.5)	24.0 (15.3)	22.2 (12.2)	33.3 (19.6)	41.6 (26.0)

Table D.1.1.a: Mean number of generations required for the SGA to solve the RR problem with 2-point crossover and a seed probability of 0.2. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	Seed pool size					
	5	**25**	**50**	**100**	**500**	**1000**
1000	495.3 *	133.4 *	168.3 *	245.9 *	421.3 *	N/A
5000	396.7 *	128.1 *	88.3	71.2	180.6 *	278.8 *
10000	345.7 *	111.1 *	74.6	70.3	99.5	177.8
50000	261.8 *	50.3	51.8	58.0	68.4	71.1
100000	292.5 *	53.5 *	46.6	55.6	75.3	78.3
500000	293.9 *	129.0	124.0	122.2	133.3	141.6

Table D.1.1.b: *Normalised* mean number of generations required for the SGA to solve the RR problem with 2-point crossover and a seed probability of 0.2. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[D.1.2] RR, Uniform Crossover

Presample	Seed pool size					
	5	**25**	**50**	**100**	**500**	**1000**
1000	624.6 (259.7) [74 runs]	535.6 (235.7) [667 runs]	553.8 (235.0) [608 runs]	586.1 (230.2) [542 runs]	628.6 (233.1) [385 runs]	N/A
5000	561.0 (249.0) [149 runs]	494.2 (237.9) [726 runs]	484.8 (236.5) [802 runs]	472.2 (240.7) [845 runs]	583.3 (225.1) [585 runs]	607.5 (223.3) [460 runs]
10000	527.3 (248.8) [217 runs]	481.1 (244.0) [749 runs]	462.5 (238.6) [828 runs]	459.2 (226.1) [857 runs]	528.1 (232.3) [771 runs]	584.3 (230.4) [573 runs]
50000	509.6 (260.8) [300 runs]	354.7 (219.6) [925 runs]	384.2 (228.6) [933 runs]	412.2 (223.6) [915 runs]	432.0 (225.0) [911 runs]	449.2 (221.8) [910 runs]
100000	523.0 (256.5) [261 runs]	329.8 (210.8) [945 runs]	294.5 (295.2) [976 runs]	356.3 (211.6) [974 runs]	432.6 (229.5) [912 runs]	423.2 (223.4) [917 runs]
500000	481.9 (261.8) [379 runs]	313.7 (209.2) [958 runs]	268.3 (185.9) [981 runs]	246.2 (174.4) [994 runs]	323.4 (203.4) [983 runs]	377.5 (218.0) [954 runs]

Table D.1.2.a: Mean number of generations required for the SGA to solve the RR problem with uniform crossover and a seed probability of 0.15. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

We used a seeding probability of 0.15 to run the SGA with uniform crossover. Although (as can be seen from tables D.1.2.a and D.1.2.b) not as effective as 2-point crossover, the SGA was still able to solve this problem/parameter combination most of the time. However, as we saw in section 5.2.2, the CGA

outperforms the SGA here. This held true for the full range of seed pool sizes we tested, and is quite expected (as explained in section 5.2.2).

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	624.8 *	535.8 *	554.0 *	586.3 *	628.8 *	N/A
5000	562.0 *	495.2 *	485.8 *	473.2 *	584.3 *	608.5 *
10000	529.3 *	483.1 *	464.5 *	461.2 *	530.1 *	586.3 *
50000	519.6 *	364.7 *	394.2 *	422.2 *	442.0 *	459.2 *
100000	543.0 *	349.8 *	314.5 *	376.3 *	452.6 *	443.2 *
500000	581.9 *	413.7 *	368.3 *	346.2 *	423.4 *	477.5 *

Table D.1.2.b: *Normalised* mean number of generations required for the SGA to solve the RR problem with uniform crossover and a seed probability of 0.15. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[D.2] Staggered Royal Road (R2)

[D.2.1] R2, 2-point Crossover

We tested the R2 problem using the approximately optimal seed probability value of 0.2. As we have already seen, the SGA performs poorly here. Table D.2.1.a is very much comparable to table C.3.1.a from the early experiments. It appears that the seed pool size begins to act to the detriment of the algorithm when it increases beyond 100, but we cannot say so with statistical certainty when so few runs completed.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	0	76	67	35	28	N/A
5000	2	84	103	119	49	28
10000	11	99	131	146	102	56
50000	17	239	227	225	188	183
100000	19	285	355	303	210	161
500000	27	342	438	523	376	274

Table D.2.1.a: Number of runs (of 1000) in which the SGA solved the R2 problem with 2-point crossover and a seed probability of 0.2. Columns represent different seed pool sizes. Rows represent different presample sizes.

[D.2.2] R2, Uniform Crossover

Uniform crossover was a little more effective, as we observed in section 5.3.2. Using a seed probability of 0.15, the SGA was at least able to solve the problem most of the time for sufficiently large presample sizes. Once again, a seed pool of 50 individuals (or slightly larger) appears to be optimal for these circumstances.

[D.3] Hierarchical Royal Road (HRR)

[D.3.1] HRR, 2-point Crossover

This is one of the problems we have observed the SGA performs well on, usually beating the mean of 268.3 generations-to-completion of the CGA, given the right parameters. Here we use seed probability 0.2. Interestingly, a smaller seed pool size is preferred here, as seen in tables D.3.1.a and D.3.1.b.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	493.5 (271.5) [133 runs]	139.1 (148.8) [978 runs]	148.8 (129.3) [996 runs]	220.7 (164.1) [991 runs]	397.0 (213.2) [940 runs]	N/A
5000	354.7 (255.4) [196 runs]	135.3 (155.8) [987 runs]	77.0 (60.3) [999 runs]	67.9 (46.7)	168.3 (110.1) [998 runs]	260.6 (162.0) [997 runs]
10000	301.5 (266.3) [292 runs]	107.2 (130.0) [985 runs]	75.9 (63.6)	65.6 (42.9)	94.1 (57.9)	166.4 (107.2)
50000	254.5 (259.2) [402 runs]	42.4 (55.7) [999 runs]	45.5 (38.6)	50.4 (33.7)	58.4 (34.6)	59.2 (34.4)
100000	195.9 (252.4) [421 runs]	32.4 (37.6)	29.0 (19.2)	39.7 (25.7)	54.6 (34.0)	59.2 (37.2)
500000	186.8 (257.9) [449 runs]	18.7 (22.2)	18.8 (11.7)	20.2 (10.8)	33.4 (18.7)	44.0 (27.7)

Table D.3.1.a: Mean number of generations required for the SGA to solve the HRR problem with 2-point crossover and a seed probability of 0.2. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	Seed pool size					
	5	**25**	**50**	**100**	**500**	**1000**
1000	493.7 *	139.3 *	302.7 *	220.9 *	397.2 *	N/A
5000	355.7 *	136.3 *	149.8 *	68.9	169.3 *	261.6 *
10000	303.5 *	109.2 *	77.9	67.6	96.1	168.4
50000	264.5 *	52.4 *	55.5	60.4	68.4	69.2
100000	215.9 *	52.4	49.0	59.7	74.6	79.2
500000	286.8 *	118.7	118.8	120.2	133.4	144.0

Table D.3.1.b: *Normalised* mean number of generations required for the SGA to solve the HRR problem with 2-point crossover and a seed probability of 0.2. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[D.3.2] HRR, Uniform Crossover

Recall that HRR with uniform crossover is the problem/parameter combination with the unusual behaviour, as discussed extensively in appendix E. We use seed probability 0.15 for these experiments.

Consider table D.3.2.a. At the upper reaches of the presample size, where the negative influence begins to show in table E.a, the larger seed pool – as large as 500 individuals – is performing better. This is exactly as expected if our hypothesis from Appendix E is correct: a sufficient number of individuals with two non-contiguous blocks or even three blocks are getting into a seed pool of size 500, where they are being crowded out in a seed pool of size 50 by individuals with a single secondary block, which the HRR algorithm classifies as “more fit” even though they are actually *less* useful for the progress of the genetic algorithm.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	558.9 (249.2) [65 runs]	535.3 (247.0) [569 runs]	541.5 (234.1) [508 runs]	587.4 (231.0) [421 runs]	633.6 (227.3) [255 runs]	N/A
5000	558.1 (260.9) [106 runs]	494.7 (235.5) [597 runs]	506.6 (236.1) [667 runs]	503.4 (241.9) [750 runs]	582.7 (228.8) [442 runs]	608.2 (221.3) [329 runs]
10000	529.1 (241.8) [187 runs]	487.7 (235.4) [651 runs]	492.3 (233.0) [728 runs]	480.6 (241.3) [754 runs]	528.4 (246.7) [630 runs]	598.3 (237.7) [483 runs]
50000	528.0 (268.8) [220 runs]	385.3 (230.4) [862 runs]	399.2 (233.4) [865 runs]	455.4 (242.6) [826 runs]	477.2 (239.8) [816 runs]	454.2 (230.9) [799 runs]
100000	561.1 (239.8) [205 runs]	376.8 (222.8) [917 runs]	338.2 (213.0) [972 runs]	388.3 (220.9) [929 runs]	456.3 (232.6) [842 runs]	462.7 (235.4) [828 runs]
300000	524.8 (253.0) [187 runs]	402.4 (238.9) [860 runs]	333.6 (211.2) [952 runs]	291.8 (193.9) [980 runs]	405.1 (230.9) [919 runs]	439.5 (236.0) [883 runs]
500000	497.9 (254.2) [223 runs]	415.5 (233.5) [822 runs]	354.5 (221.9) [936 runs]	309.3 (203.7) [982 runs]	367.3 (215.1) [954 runs]	425.3 (233.5) [906 runs]

Table D.3.2.a: Mean number of generations required for the SGA to solve the HRR problem with uniform crossover and a seed probability of 0.15. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ). Entries of interest (cases where the mean *increased* or the number of runs completing *decreased* with a larger presample) are highlighted in yellow.

[D.4] One-Sum (OS)

[D.4.1] OS, 2-point Crossover

The OS problem behaves similarly to previous experiments with the seed pool size. We used a seeding probability of 0.15, and found that a seed pool around 50 was the "magic number". See tables D.4.1.a and D.4.1.b below for the results. The optimal presample size (after normalisation) for this problem is very small: in the 5000-10000 range. This is likely because the algorithm is able to build up more fitness for this simple problem in a few generations than it is by increasing the presample by the amount equating to those few generations.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	76.1 (77.3) [874 runs]	36.2 (17.9)	37.2 (17.2)	41.0 (18.4)	55.5 (23.1)	N/A
5000	60.4 (63.1) [923 runs]	28.6 (13.6)	27.9 (12.0)	30.5 (12.4)	38.8 (16.8)	45.4 (19.1)
10000	55.9 (63.2) [934 runs]	25.6 (11.9)	26.6 (11.7)	27.3 (11.5)	33.8 (14.5)	38.0 (16.0)
50000	47.3 (55.1) [947 runs]	20.8 (8.7)	20.7 (7.8)	21.5 (8.2)	26.7 (10.5)	29.0 (11.3)
100000	39.4 (42.6) [949 runs]	19.2 (7.7)	19.0 (5.6)	19.9 (7.2)	23.6 (8.8)	26.0 (9.9)
500000	31.2 (36.6) [980 runs]	16.3 (6.1)	16.3 (7.0)	17.2 (5.9)	19.6 (6.7)	21.2 (8.0)

Table D.4.1.a: Mean number of generations required for the SGA to solve the OS problem with 2-point crossover and a seed probability of 0.15. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	76.3 *	36.4	37.4	41.2	55.7	N/A
5000	61.4 *	29.6	28.9	31.5	39.8	46.4
10000	57.9 *	27.6	28.6	29.3	35.8	40.0
50000	57.3 *	30.8	30.7	31.5	36.7	39.0
100000	59.4 *	39.2	39.0	39.9	43.6	46.0
500000	131.2 *	116.3	116.3	117.2	119.6	121.2

Table D.4.1.b: *Normalised* mean number of generations required for the SGA to solve the OS problem with 2-point crossover and a seed probability of 0.15. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[D.4.2] OS, Uniform Crossover

The variant using uniform crossover was similarly unsurprising. We used seed probability 0.1. The normalised data is presented in table D.4.2.a; notice again that the optimal presample size for this simple problem is smaller than we have seen so far.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	15.5 *	13.6	14.3	15.3	18.8	N/A
5000	13.8 *	12.7	13.4	13.8	15.9	17.3
10000	14.5 *	13.6	13.9	14.2	15.7	16.8
50000	21.0 *	20.6	20.7	21.2	22.0	22.9
100000	30.7 *	30.2	30.4	30.7	31.6	32.1
500000	109.9 *	109.5	109.7	110.1	110.8	111.1

Table D.4.2.a: *Normalised* mean number of generations required for the SGA to solve the OS problem with uniform crossover and a seed probability of 0.1. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

[D.5] Hierarchical If-And-Only-If (HIFF), 2-point Crossover

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	590.7 (370.7) [12 runs]	508.5 (278.8) [75 runs]	549.0 (265.9) [125 runs]	508.5 (257.4) [75 runs]	439.1 (203.8) [16 runs]	N/A
5000	385.8 (263.5) [37 runs]	506.7 (280.0) [218 runs]	493.9 (264.0) [292 runs]	506.7 (272.4) [218 runs]	523.7 (263.6) [98 runs]	456.2 (283.1) [44 runs]
10000	399.0 (293.9) [41 runs]	503.3 (276.6) [309 runs]	474.0 (267.8) [358 runs]	503.3 (271.2) [309 runs]	537.4 (270.5) [140 runs]	512.4 (268.2) [94 runs]
50000	321.8 (271.6) [60 runs]	323.6 (267.5) [635 runs]	389.2 (273.3) [566 runs]	441.1 (277.8) [552 runs]	489.8 (260.3) [346 runs]	506.5 (257.3) [252 runs]
100000	270.0 (263.8) [91 runs]	252.8 (265.0) [824 runs]	380.9 (278.7) [697 runs]	403.0 (271.5) [687 runs]	495.7 (267.6) [452 runs]	484.0 (262.9) [313 runs]
500000	255.6 (259.6) [153 runs]	286.8 (243.7) [941 runs]	269.8 (236.4) [929 runs]	286.8 (236.9) [941 runs]	407.0 (266.7) [746 runs]	433.0 (271.8) [618 runs]

Table D.5.a: Mean number of generations required for the SGA to solve the HIFF problem with 2-point crossover and a seed probability of 0.3. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

Recall that the CGA was essentially unable to solve the HIFF problem. Using a seed probability of 0.3, we achieved the results displayed in table D.5.a above. Once again, we find the algorithm succeeding the most on a seed pool size around 50.

We did not attempt the uniform variant again, given the lack of successes with our 20000-generation tests there. There was no plausible reason why a smaller or larger seed pool size should suddenly increase the success rates.

[D.6] Deceptive Trap (DT), 2-point Crossover

Recall that the CGA fails on the DT problem, and that the SGA is rather successful. We ran experiments using a seed probability of 0.175. Refer to table D.6.a for the results. 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. A seed pool of 100 is statistically significantly better than 50 for presample size 100000, and is significantly better than either 500 or 25 for presample sizes of 5000 and higher.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	308.0 (275.3) [3 runs]	533.2 (244.7) [440 runs]	542.5 (223.9) [656 runs]	578.9 (215.3) [766 runs]	721.0 (191.9) [396 runs]	N/A
5000	304.6 (164.0) [5 runs]	424.6 (239.3) [750 runs]	406.7 (215.6) [924 runs]	400.5 (200.0) [947 runs]	520.7 (212.5) [914 runs]	623.6 (205.7) [812 runs]
10000	258.8 (191.4) [17 runs]	379.0 (225.8) [796 runs]	343.1 (194.0) [950 runs]	335.7 (181.3) [985 runs]	427.2 (192.2) [978 runs]	516.7 (206.4) [931 runs]
50000	350.7 (250.5) [30 runs]	277.5 (196.2) [912 runs]	227.6 (147.2) [988 runs]	227.0 (137.5) [999 runs]	285.1 (148.5) [998 runs]	336.9 (162.6)
100000	263.2 (236.4) [47 runs]	266.2 (182.1) [965 runs]	200.6 (141.3) [997 runs]	185.7 (109.0) [999 runs]	237.3 (126.7)	282.5 (130.7)
500000	228.7 (186.8) [102 runs]	164.8 (130.7) [990 runs]	132.7 (100.5)	128.1 (78.5)	162.1 (86.2)	184.8 (94.2)

Table D.6.a: Mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed probability of 0.175. Columns represent different seed pool sizes. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ).

The utility of larger seed pools for this function can also be seen in the normalised data (table D.6.b). A plausible explanation for this lies in the nature of the building blocks. For all the Royal Road variants, the (primary) blocks are of length eight, and there are eight of them; when the proportion of the seed pool size to the presample size rises above a certain level, the seed pool will begin to fill with blocks containing few or even no blocks. For DT, though, the blocks are of length five. This means that the

chance a random individual contains at least one correct block is much higher than for the Royal Road variants. Therefore for a given presample size, the presample generated for DT will be able to fill a larger seed pool with reasonably fit candidates than any Royal Road variant can.

Of course, when increasing the seed pool size but keeping the presample size stable, the DT will be filling the seed pool with more and more second-rate, i.e. deceptive, blocks. This must ameliorate the ability of large seed pool sizes to be viable here, but does not appear to be too significant a disadvantage within the seed pool sizes we tested.

Presample	**Seed pool size**					
	5	**25**	**50**	**100**	**500**	**1000**
1000	308.2 *	533.4 *	542.7 *	579.1 *	721.2 *	N/A
5000	305.6 *	425.6 *	407.7 *	401.5 *	521.7 *	624.6 *
10000	260.8 *	381.0 *	345.1 *	337.7 *	429.2 *	518.7 *
50000	360.7 *	287.5 *	237.6 *	237.0 *	295.1 *	346.9
100000	283.2 *	286.2 *	220.6 *	205.7 *	257.3	302.5
500000	328.7 *	264.8 *	232.7	228.1	262.1	284.8

Table D.6.b: *Normalised* mean number of generations required for the SGA to solve the DT problem with 2-point crossover and a seed probability of 0.175. Columns represent different seed pool sizes. Rows represent different presample sizes. An asterisk * indicates that not all of the 1000 runs completed within 1000 generations; entries where fewer than 90% of the runs completed or where the presample size is equal to the seed pool size are greyed out. Other entries are coloured according to the data format given in Appendix C, section C.1.

We did not attempt the uniform variant again, given the failure to solve it in 20000 generations. There was no plausible reason why a smaller or larger seed pool size should suddenly increase the success rates.

Appendix E: Incongruous Results with HRR and Uniform Crossover

We look here in more detail at the case of the SGA running on the HRR problem with uniform crossover.

From the perspective of either normalised *or* raw data, the optimal values were a presample size of 100000 – 200000, with a seed probability of 15%. This is similar to the trend for RR and R2 using uniform crossover.

Observe the results presented in table E.a. Note the strange discrepancy: at some point, increasing the presample size *causes the performance of the seeding algorithm to worsen*. This is a statistically significant change for the large differences such as 400000 to 600000 with seed probability 0.1, and 300000 to 600000 with seed probability 0.15.

Presample	**Seed probability**				
	0.1	**0.15**	**0.2**	**0.25**	**0.3**
1000	657.6 [278 runs]	564.1 [504 runs]	- [0 runs]	- [0 runs]	- [0 runs]
10000	593.2 [416 runs]	498.5 [732 runs]	370.0 [2 runs]	- [0 runs]	- [0 runs]
50000	539.9 [618 runs]	406.9 [891 runs]	532.8 [10 runs]	- [0 runs]	- [0 runs]
100000	462.3 [758 runs]	339.0 [956 runs]	536.1 [59 runs]	- [0 runs]	- [0 runs]
200000	459.0 [798 runs]	327.7 [965 runs]	573.7 [60 runs]	- [0 runs]	- [0 runs]
300000	454.3 [794 runs]	338.9 [953 runs]	546.2 [45 runs]	- [0 runs]	- [0 runs]
400000	438.8 [797 runs]	356.9 [963 runs]	453.8 [25 runs]	- [0 runs]	- [0 runs]
500000	473.4 [782 runs]	359.5 [938 runs]	520.6 [19 runs]	- [0 runs]	- [0 runs]
600000	475.4 [753 runs]	363.2 [932 runs]	621.7 [13 runs]	- [0 runs]	- [0 runs]

Table E.a: Mean number of generations required for the SGA to solve the HRR problem with uniform crossover and a seed pool of 50. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ). Entries of interest (cases where the mean *increased* or the number of runs completing *decreased* with a larger presample) are highlighted in yellow.

Nothing in our initial model accounted for this behaviour. At *worst*, increasing the presample should have no effect on performance. We did have one plausible direction for our inquiry, however: this is the *Hierarchical* Royal Road. A larger presample will typically mean 'better' individuals in the sense of

individuals with more *contiguous* blocks, rather than individuals with *more* blocks. Furthermore, the Hierarchical Royal Road is prone to *hitchhiking* of poor genetic material on individuals with higher order blocks. [Mitchell *et al*, 1992]

Statistically speaking, with a presample in the hundreds of thousands, there should always be enough 2-block individuals to fill the pool. Increasing the presample size after this point may cause the seed pool to fill with individuals with two contiguous blocks. The 'hierarchical' structure of the problem does not *in and of itself* make it easier to solve, as [Mitchell *et al*, 1992] demonstrated. It also makes no difference whatsoever to the efficacy of *uniform* crossover whether two blocks are contiguous or distant in the bit string. Therefore the only effect when we increase the presample at this point is that the seed pool is filling with individuals with a secondary block, made up of two contiguous blocks – of which there are only four possible variants – instead of individuals with two blocks randomly distributed – of which there are 28 possible variants. So what we are seeing could in fact be a *loss of diversity*.

This is only conjecture, so we next conducted experiments to test the hypothesis. First we look at the actual composition of the seed pool, and then we artificially "stack" the seed pool.

[E.1] Repeating Results with More Data Collected

Presample	**Seed probability**					
	0.075	**0.1**	**0.125**	**0.15**	**0.175**	**0.2**
200000	518.7 (272.0) [568 runs]	452.8 (272.4) [765 runs]	382.0 (238.0) [934 runs]	326.5 (213.3) [970 runs]	421.6 (247.7) [683 runs]	515.6 (264.6) [58 runs]
300000	530.9 (264.0) [543 runs]	454.0 (263.0) [807 runs]	376.9 (232.2) [921 runs]	339.1 (212.5) [953 runs]	442.0 (260.2) [598 runs]	506.0 (281.4) [42 runs]
400000	523.9 (266.6) [520 runs]	435.4 (255.9) [795 runs]	386.8 (231.3) [912 runs]	345.3 (210.4) [960 runs]	458.3 (246.9) [562 runs]	515.3 (287.4) [33 runs]
500000	528.1 (269.9) [569 runs]	473.4 (255.0) [773 runs]	399.0 (234.3) [924 runs]	354.7 (215.5) [941 runs]	465.2 (257.2) [523 runs]	532.4 (297.2) [25 runs]
600000	514.9 (263.3) [553 runs]	489.1 (261.4) [794 runs]	393.4 (229.6) [923 runs]	353.2 (216.6) [939 runs]	451.1 (258.3) [473 runs]	467.1 (218.9) [19 runs]
700000	529.9 (256.7) [530 runs]	471.1 (252.5) [749 runs]	413.3 (232.2) [912 runs]	372.2 (219.9) [919 runs]	477.3 (258.3) [415 runs]	455.6 (250.2) [13 runs]
800000	525.8 (266.1) [501 runs]	480.2 (255.4) [766 runs]	416.9 (236.1) [921 runs]	389.4 (230.3) [918 runs]	493.8 (259.7) [395 runs]	574.5 (388.7) [6 runs]

Table E.1.a: Repeated result for the mean number of generations required for the SGA to solve the HRR problem with uniform crossover, a seed pool of 50, and a wider range of seed probabilities. Columns represent different seed probabilities. Rows represent different presample sizes. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ). Entries of interest (cases where the mean increased or the number of runs completing decreased with a larger presample) are highlighted in yellow.

We first made sure the incongruous results were repeatable. This was also an opportunity to collect more data. We set up an experiment using the same parameters, and storing more data about the composition of the seed pool. We looked at the larger presample sizes, with low seed probabilities, where the problem behaviour was occurring. We also looked at intermediate seed probabilities (0.075, 0.125, and 0.175) to get greater coverage of the problem, and tested presample sizes of 700000 and 800000 to discover how the trend continued.

The repeated results are given in table E.1.a above. We observe a statistically significant negative difference between presample sizes of 300000 and 800000, for each seed probability, other than at the extremes of 0.075 and 0.2.

[E.2] Block-level View of the Seed Pool

We now look at the data broken down into:

- The mean number of individuals in the seed pool with 1, 2, 3, or 4 elementary blocks.
- The mean number of individuals in the seed pool with 0, 1, or 2 secondary blocks.
- The mean number of individuals in the seed pool with at least 1 tertiary block.
- The total number of each block b_i, for $i = 1$ to 8.

For the HRR problem, by 'elementary block', we mean a block of eight continuous bits set to one, in one of eight specific positions in the bit string. By 'secondary block', we mean two contiguous blocks; specifically, the first quarter, second quarter, third quarter or last quarter of the bit string. By 'tertiary block', we mean either the first or second half of the bit string. The data is given in tables E.2.a, E.2.b, E.2.c, and E.2.d.

In all runs, in all experiments, each individual in the seed pool had more than one elementary block. In the average case, there tended to be ~2 individuals with three primary blocks, and the other ~48 each had two primary blocks (see tables E.2.a and E.2.b). This is exactly as statistically predicted[30]. The mean number individuals in the seed pool that contained four elementary blocks never exceeded 0.02.

Presample	**Seed probability**					
	0.075	**0.1**	**0.125**	**0.15**	**0.175**	**0.2**
200000	49.3	49.3	49.4	49.3	49.3	49.3
300000	49.0	49.0	49.0	49.0	49.0	49.0
400000	48.6	48.7	48.6	48.6	48.7	48.7
500000	48.3	48.4	48.3	48.3	48.5	48.3
600000	48.0	48.0	48.0	48.0	47.9	48.1
700000	48.0	48.0	47.9	48.0	47.9	47.9
800000	48.2	48.1	48.1	48.1	48.2	48.1

Table E.2.a: mean number of individuals in the seed pool with exactly two elementary blocks, for the SGA on the HRR problem with uniform crossover and a seed pool of 50.

30 See for example the analysis in section 5.2.1.

Presample	**Seed probability**					
	0.075	**0.1**	**0.125**	**0.15**	**0.175**	**0.2**
200000	0.6	0.7	0.6	0.7	0.7	0.7
300000	1.0	1.0	1.0	1.0	1.0	1.0
400000	1.4	1.3	1.4	1.4	1.3	1.3
500000	1.7	1.6	1.7	1.7	1.5	1.7
600000	1.9	2.0	2.0	2.0	2.0	1.9
700000	2.0	2.0	2.1	2.0	2.0	2.1
800000	1.8	1.9	1.9	1.9	1.8	1.8

Table E.2.b: mean number of individuals in the seed pool with exactly three elementary blocks, for the SGA on the HRR problem with uniform crossover and a seed pool of 50.

Presample	**Seed probability**					
	0.075	**0.1**	**0.125**	**0.15**	**0.175**	**0.2**
200000	37.6	37.6	37.8	37.8	37.9	37.9
300000	31.7	31.7	31.8	31.5	31.7	31.7
400000	25.4	25.8	25.4	25.7	25.4	25.9
500000	19.3	19.6	19.5	19.4	19.6	19.7
600000	13.2	13.3	13.7	13.7	13.2	13.5
700000	7.7	7.6	7.6	7.4	7.9	7.8
800000	3.3	3.2	3.3	3.4	3.5	3.5

Table E.2.c: mean number of individuals in the seed pool with no secondary blocks, for the SGA on the HRR problem with uniform crossover and a seed pool of 50.

Presample	**Seed probability**					
	0.075	**0.1**	**0.125**	**0.15**	**0.175**	**0.2**
200000	12.4	12.4	12.2	12.2	12.1	12.1
300000	18.3	18.3	18.2	18.5	18.3	18.3
400000	24.6	24.2	24.6	24.3	24.6	24.1
500000	30.7	30.4	30.5	30.6	30.4	30.3
600000	36.8	36.7	36.3	36.3	36.8	36.5
700000	42.3	42.4	42.4	42.6	42.1	42.2
800000	46.7	46.8	46.7	46.6	46.5	46.5

Table E.2.d: mean number of individuals in the seed pool with exactly one secondary block, for the SGA on the HRR problem with uniform crossover and a seed pool of 50.

The first evidence supporting our hypothesis about the SGA performance can be seen in tables E.2.c and E.2.d. The increase in the mean number of seed individuals with secondary blocks when increasing the presample size is huge: 34 more individuals over the tested range. This is almost a threefold increase in

the number of individuals with one of the four possible "two elementary blocks arranged in a secondary block" schemata. Correspondingly, the number of individuals with one of the 28 possible "any two elementary blocks" schemata is reduced to about one-tenth of its former state. This represents a massive loss of schema variability.

The mean number of individuals with a higher fitness was small enough that we can discount its influence: 0 or < 0.01 individuals having two secondary blocks were equally common; only 11 of the 42 experiments of 1000 runs recorded the appearance of *any* tertiary blocks whatsoever.

As mentioned, for each experiment we also tracked the mean number of each block b_i, for $i = 1$ to 8, occurring in the seed pool. Any significant deviation in the frequency of some particular block b_x would have indicated that the decreasing performance on this problem with increasing presample size was due to systematic or experimenter error.

Fortunately, we were able to rule this out: the frequency of each block in the seed pool was typically equal to the mean over all blocks, ± 0.1, and no block b_x deviated from the mean frequency any more than the other blocks.

[E.3] Results of Extra Data Collection (Block Counting)

Our hypothesis about the detrimental effect of secondary blocks remained intact. We next designed an experiment to directly test it.

Consider our HRR implementation. The maximum fitness for an individual is 192, and the maximum number of blocks is 14. Each 8-bit primary block grants +8 fitness; each 16-bit secondary block grants +16 fitness; each 32-bit secondary block grants +32 fitness. The key point to keep in mind is that it is possible to have a higher fitness with fewer elementary blocks present in a string: an individual with four non-adjacent primary blocks has fitness 32, but an individual with two adjacent primary blocks forming a secondary block, plus one other primary block, has fitness 40.

We define *hierarchical schemata* for use in the following results. The phrase "*x*_p_*y*_s_*z*_t", or (partial) variations thereof, shall refer to individuals with schemata of the form: *x* primary blocks, *y* secondary blocks, and *z* tertiary blocks. For example, "2_s" refers to individuals with exactly two secondary blocks and no other blocks; "1_t_2_p" refers to individuals with one tertiary block, two other primary blocks, and no other blocks.

Our experiment proceeds as follows. We "stack" the seed pool by filling it with random variants meeting a particular hierarchical schema, instead of using a presample. Thus a "3_p" seed pool might look like figure E.3.a, below.[31]

31 Note that what may appear to be higher-order blocks in the third and sixth individuals do *not* correspond to what the HRR fitness function refers to as secondary blocks: they do not fall within the "quarters" of the bit string.

11111111xxxxxxxxxxxxxxxxxxxxxxxx11111111xxxxxxxx11111111xxxxxxxx
xxxxxxxxxxxxxxxx11111111xxxxxxxx11111111xxxxxxxx11111111xxxxxxxx
xxxxxxxx1111111111111111xxxxxxxx11111111xxxxxxxxxxxxxxxxxxxxxxxx
11111111xxxxxxxx11111111xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx11111111
Xxxxxxxx11111111xxxxxxxxxxxxxxxx11111111xxxxxxxx11111111xxxxxxxx
xxxxxxxxxxxxxxxx11111111xxxxxxxxxxxxxxxx1111111111111111xxxxxxxx
...etc

Figure E.3.a: example of a "3_p" seed pool. All individuals precisely meet the "3_p" hierarchical schema, which has 24 fitness. Bits marked as an 'x' are randomly assigned the value 0 or 1 (but *not* in such a way that they form additional blocks).

		Seed probability		
	HRR fitness	**0.1**	**0.15**	**0.2**
1_p	**8**	638.4 (223.1) [402 runs]	519.9 (244.1) [666 runs]	848.5 (118.8) [4 runs]
2_p	**16**	445.8 (261.8) [803 runs]	303.7 (204.7) [977 runs]	521.1 (281.9) [127 runs]
3_p	**24**	224.9 (207.1) [984 runs]	110.2 (90.0)	259.6 (212.7) [942 runs]
4_p	**32**	63.5 (80.2)	38.8 (34.5)	47.1 (31.1)
1_s	**32**	512.1 (251.0) [745 runs]	401.4 (232.1) [901 runs]	748.6 (247.7) [7 runs]
1_s_1_p	**40**	269.0 (220.0) [970 runs]	181.1 (130.9) [999 runs]	476.2 (271.8) [341 runs]
1_s_2_p	**48**	77.5 (99.0)	55.8 (47.1)	114.5 (101.7)
1_s_3_p	**56**	19.3 (22.0)	15.8 (9.9)	19.7 (10.6)
2_s	**64**	164.0 (130.3) [999 runs]	131.249 (82.8)	417.2 (265.7) [345 runs]
2_s_1_p	**72**	31.1 (32.3)	27.2 (18.8)	47.3 (37.2)
2_s_2_p	**80**	8.8 (3.6)	8.2 (2.6)	8.5 (2.8)
1_t	**96**	195.1 (148.1) [996 runs]	186.7 (126.1) [998 runs]	505.9 (274.0) [228 runs]
1_t_1_p	**104**	41.0 (42.2)	39.8 (29.2)	81.8 (72.1) [999 runs]
1_t_2_p	**112**	9.7 (4.1)	9.4 (3.7)	9.9 (4.7)
1_t_1_s	**128**	11.4 (7.0)	11.2 (5.7)	13.6 (11.8)
1_t_1_s_1_p	**136**	4.8 (1.3)	3.9 (1.1)	3.3 (1.1)

Table E.3.b: Mean number of generation for the SGA to solve the HRR problem with uniform crossover after "stacking" the seed pool with various schemata. Columns represent different seed probabilities. Rows represent different stacked seed pools. The comment [*x* runs] indicates that only *x* of 1000 runs completed within 1000 generations. Standard deviations are given in parentheses (σ). Entries of interest are highlighted.

There are 16 possible hierarchical schemata for the HRR problem. For each of these, we ran the SGA for 1000 runs, using 5-tournament selection, for seed probabilities 0.1, 0.15, and 0.2. Table E.3.b, above, gives the results of these experiments. Note that the rows are laid out according to the fitness that the HRR fitness function assigns to the *x*_p_*y*_s_*z*_t hierarchical schema, strictly increasing. However, it is clearly *not* the case that running the SGA with a seed pool stacked with individuals of higher fitness always improves performance in this situation.

Consider the rows for the hierarchical schemata "2_p" and "1_s", highlighted blue and yellow respectively. We saw in section E.2 that a smaller presample essentially fills the seed pool with instances of "2_p", which are replaced with "1_s" as the presample grows in size, because HRR considers these to have higher fitness.

As we can see from table E.3.b, the SGA with uniform crossover does better using a seed pool stacked with "2_p" than it does with a seed pool stacked with "1_s". There is an extremely statistically significant difference for the two smaller tested probabilities (P values < 0.0001). The drop from 127 completing to 7 completing for 0.02 is also compelling (although we cannot make any statistically valid claims there).

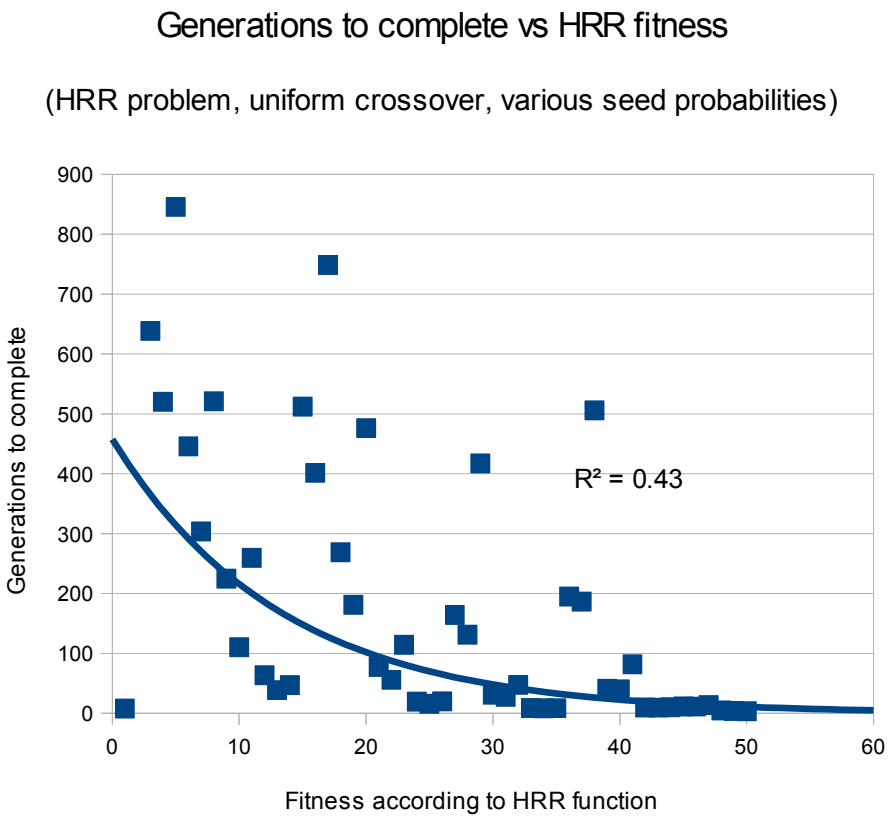

Figure E.3.c: Mean number of generations for the SGA to solve the HRR problem with uniform crossover, charted against the fitness of the seed individuals according to the HRR fitness function.

The rows in table E.3.b coloured salmon and green are for "3_p" and "1_s_1_p" respectively. Recall from section E.2 that at the highest sizes tested, a few "1_s_1_p" individuals also appeared in the pool. These

have a higher assigned fitness than "3_p", and would crowd them out. However, from table E.3.b we can see that "3_p" individuals are more useful than "1_s_1_p" individuals (the difference is statistically significant at the seeding probabilities 0.15 and 0.2).

This is all evidence supporting our theory. As a final test, we consider the relationships between the mean generations to completion, the HRR fitness score, and the number of 1s belonging to fixed blocks in seed individuals.

Figure E.3.c charts the mean number of generations it takes the SGA to solve the HRR problem with uniform crossover against the fitness of seed pool individuals. Figure E.3.d charts the same mean against the number of 1s in blocks of the seed individuals. The data for both charts comes from table E.1.a.

We use curve-fitting functions to find the line of best fit in each chart. The coefficient of determination R^2 is considerably higher for figure E.3.d. This better fit suggests that the performance of the SGA, in terms of mean number of generations for this problem, correlates more closely to the number of 1s within blocks of seed individuals than it does with the HRR fitness of the seed individuals.

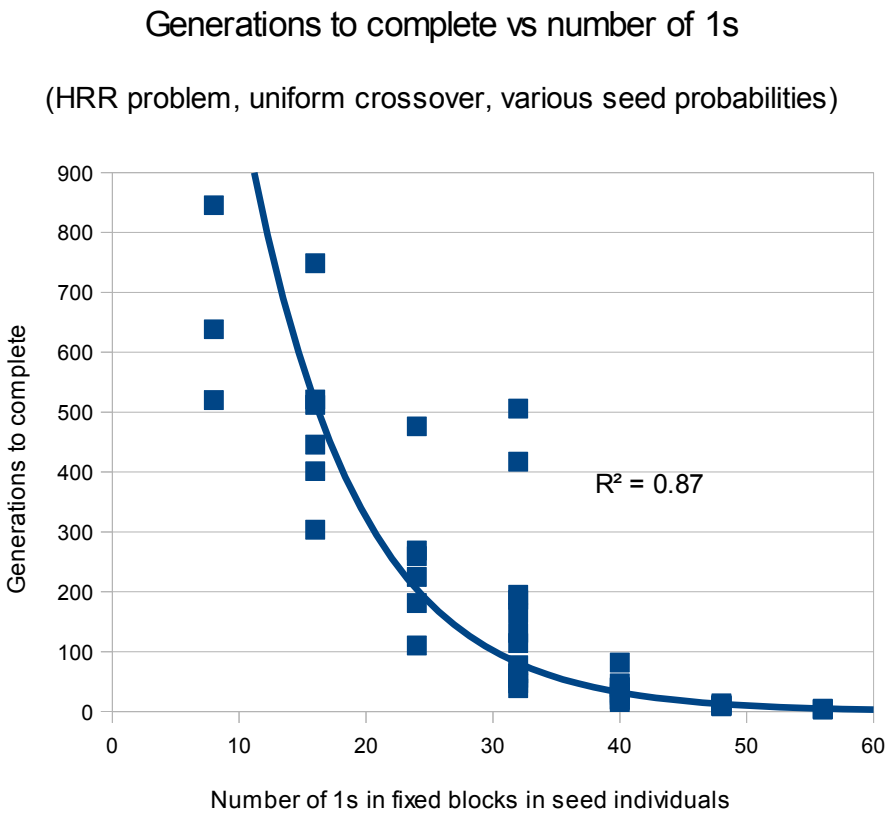

Figure E.3.d: Mean number of generations for the SGA to solve the HRR problem with uniform crossover, charted against the number of 1-bits tied up in blocks in the seed individuals.

This correlation is important, because when individuals with one secondary block are present in the presample, the SGA will *always* choose them over individuals with two or even three non-adjacent primary blocks, during the process of seed pool population. This is simply because individuals with a secondary block have a higher fitness according to the HRR fitness function. In fact, they will also crowd out individuals with three discrete primary blocks! And according to the charts and figures presented in this section, the SGA performs better on those individuals with several non-adjacent primary blocks.

Refer back to section C.2.1. The chance of generating an individual with three blocks is 1.93358×10^{-5}. At a presample size of 800000, there should be 15.5 such individuals generated. Now, there are 46 ways of arranging three primary blocks in the bit string. 24 of these form a secondary block and a primary block. So with a presample size of 800000, there should be 15.5 x 22/46, or 7.4, individuals generated with three non-adjacent primary blocks.

According to table E.2.b, only 1.9 of these individuals make it into the seed pool, on average: roughly a quarter. Their "1_s_1_p" cohorts are selected for the seed pool, which is good, but the rest of the pool tends to be filled with individuals with a single secondary block instead of the remaining "3_p" individuals. This would appear to be the core of our problem.

We would expect that as the presample got much larger, the SGA would sway back to being more efficient, as "1_s_1_p" begins to appear in numbers large enough to dominate the seed pool, to the exclusion of "1_s". However, this could not happen within 'reasonable' presample sizes.

[E.4] Conclusions for the HRR 'Anomaly'

The SGA works well on the HRR problem with 2-point crossover. Increasing the presample size increases the total number of primary blocks in the population, which is good, or increases the total number of secondary blocks – and therefore the number of *contiguous* blocks – in the population. This too is good, even though it may decrease the total number of primary blocks, because 2-point crossover *is better able to pass multiple blocks on to the next generation if they are adjacent*.

Uniform crossover, however, does not "care" about the position of blocks in a bit string; a stronger presence of "1_s" individuals in the seed pool at the expense of "2_p" individuals is *bad*, because of a loss of diversity, and the hitchhiking of inferior genetic material. [Mitchell *et al*, 1992]. Even worse, it comes at the expense of "3_p" individuals.

Even though the problem is "hierarchical", its GA-easiness correlates more closely with the total number of 1s in blocks than it does with the total number of higher-order blocks. Since at some point, increasing the presample increases the proliferation of "1_s" individuals at the expense of "2_p" and "3_p", the performance of the SGA suffers correspondingly at that point.

[9.] References

Cohen, P.R., Kim, J.B.: A Bootstrap Test For Comparing Performance Of Programs When Data Are Censored, And Comparisons To Etzioni's Test. University of Massachusetts (1993)

Darwen, P., Xin, Y.: Every niching method has its niche: Fitness sharing and implicit sharing compared. In Hans-Michael, V., Werner, E., Ingo, R., Schwefel, H., eds.: Parallel Problem Solving from Nature - PPSN IV: Lecture Notes in Computer Science. Berlin: Springer (1996) 398-407

De Jong, K. A.: An analysis of the behavior of a class of adaptive genetic adaptive systems. Doctoral dissertation, University of Michigan (1975)

Etzioni, O., Etzioni, R.: Statistical Methods for Analyzing Speedup Learning Experiments. In: Machine Learning. (1994) 333-347

Forrest, S., Mitchell, M.: Relative Building Block Fitness and the Building Block Hypothesis. In Witley, L. D., ed.: Foundation of Genetic Algorithms 2, San Mateo: Morgan Kauffman (1993)

Giguere, P., Goldberg, D. E.: Population sizing for optimum sampling with genetic algorithms: A case study of the Onemax problem. In Proceedings of the Third Anual Genetic Programming Conference 98, (1998) 496–503

Goldberg, D.E.: Simple genetic algorithms and the minimal, deceptive problem. In: Genetic Algorithms and Simulated Annealing. Morgan Kaufmann (1987) 74-88

Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989)

Holland, J.H.: Adaptation in Natural and Artificial Systems. The University of Michigan Press (1975)

Koza, J. R.: Survey of genetic algorithms and genetic programming. In: Microelectronics Communications Technology Producing Quality Products Mobile and Portable Power Emerging Technologies: record of WESCON '95, (1995) 58-59

Luke, S., Balan, G., Panait, L.: Population Implosion in Genetic Programming. In Cantú-Paz, E., *et al*., eds.: Genetic and Evolutionary Computation - GECCO 2003, Lecture Notes in Computer Science 2724, Springer Berlin / Heidelberg (2003) 206

Mitchell, M., Forrest, S.: Fitness Landscapes: Royal Road Functions. In Back, T., Fogel, D., Michalewicz, Z., eds.: Handbook of Evolutionary Computation, 1997, Oxford University Press (1997)

Mitchell, M., Holland, J. H., Forrest, S.: When will a Genetic Algorithm Outperform Hill Climbing? In Cowan, J. D., Tesauro, G., Alspector, J., eds.: Advances in Neural Information Processing Systems, Vol. 6 (1994) 51-58

Mitchell, M., Forrest, S., Holland, J.H.: The royal road for genetic algorithms: Fitness landscapes and GA performance. In Varela, F.J., Bourgine, P., eds.: Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, 1991, MIT Press (1992) 245-254

Mühlenbein, H., Paab, G.: From recombination of genes to the estimation of distributions I. Binary parameters. In Eiben, A., Bäck, T., Shoenauer, M., Schwefel, H., eds.: Parallel Problem Solving from Nature - PPSN IV, Berlin: Springer Verlag (1996) 178-187

Pelikan, M., Goldberg, D. E.: Hierarchical Problem Solving by the Bayesian Optimisation Algorithm. IlliGAL Report No. 2000002, Illinois Genetic Algorithms Laboratory, University of Illinois (2000)

Pétrowski, A.: A Clearing Procedure as a Niching Method for Genetic Algorithms. In Proceedings of IEEE International Conference on Evolutionary Computation, 1996, Inst. Nat. des Telecommun., Evry, France (1996) 798-803

Sarma, J., De Jong, K.: An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In Bäck, T., ed.: Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kaufmann (1997) 181-186

Sehitoglu, O. T., Ucoluk, G.: A Building Block Favoring Reordering Method for Gene Positions in Genetic Algorithms. In Spector, L., *et al.*, eds.: Proceedings of the Genetic and Evolutionary Computation Conference July 2001 (2001) 571-575

Skinner, C.: On the discovery, selection and combination of building blocks in evolutionary algorithms. PhD Thesis, University of Auckland (2009)

Srinivas, M., Patnaik, L.M.: Genetic algorithms: a survey. Computer 27:6 (1994) 17-26

Thierens, D., Goldberg, D. E.: Convergence Models of Genetic Algorithm Selection Schemes. Parallel Problem Solving from Nature 3 (1994) 119-129.

Pollack, J.B., Watson, R.A.: Recombination without respect: Schema combination and disruption in genetic algorithm crossover. In *et al*, D.W., eds.: Proc. GECCO-2000, Morgan Kaufmann (2000) 112-119

Whitley, D., Rana, S., Heckendorn, R.: The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. In Journal of Computing and Information Technology 7 (1998) 33-47