for GAs to optimize. He concludes that deception is neither necessary nor sufficient to cause difficulties for GAs, and that its relevance to the study of GAs remains to be demonstrated.

There is nothing to indicate that the features listed above harm search performance of GAs; they only demonstrate the danger of drawing conclusions about the expected behavior of GAs from the static average fitnesses of schemas. Instead, a more dynamic approach is needed that takes into account the biases introduced by selection at each generation. Such approaches are described in the next several sections.

4.2 ROYAL ROADS

Royal Road Functions

The Schema Theorem, by itself, addresses the positive effects of selection (allocating increasing samples of schemas with observed high performance) but only the negative aspects of crossover—that is, the extent to which it disrupts schemas. It does not address the question of how crossover works to recombine highly fit schemas, even though this is seen by many as the major source of the search power of genetic algorithms. The Building Block Hypothesis states that crossover combines short, observed high-performance schemas into increasingly fit candidate solutions, but does not give any detailed description of how this combination occurs.

To investigate schema processing and recombination in more detail, Stephanie Forrest, John Holland, and I designed a class of fitness landscapes, called Royal Road functions, that were meant to capture the essence of building blocks in an idealized form (Mitchell, Forrest, and Holland 1992; Forrest and Mitchell 1993b; Mitchell, Holland, and Forrest 1994).

The Building Block Hypothesis suggests two features of fitness landscapes that are particularly relevant to genetic algorithms: the presence of short, low-order, highly fit schemas; and the presence of intermediate "stepping stones"—intermediate-order higher-fitness schemas that result from combinations of the lower-order schemas and that, in turn, can combine to create even higher-fitness schemas.

A fitness function (Royal Road R_1) that explicitly contains these features is illustrated in figure 4.1. R_1 is defined using a list of schemas s_i. Each s_i is given with a coefficient c_i. The fitness $R_1(x)$ of a bit string x is defined as

$$R_1(x) = \sum_i c_i \delta_i(x), \quad \text{where } \delta_i(x) = \begin{cases} 1 & \text{if } x \in s_i \\ 0 & \text{otherwise.} \end{cases}$$

For example, if x is an instance of exactly two of the order-8 schemas, $R_1(x) = 16$. Likewise, $R_1(111\cdots1) = 64$.

s_1 = 11111111**********...**********; $c_1 = 8$
s_2 = ********11111111**********...**********; $c_2 = 8$
s_3 = ****************11111111**********...**********; $c_3 = 8$
s_4 = ************************11111111**********...**********; $c_4 = 8$
s_5 = ********************************11111111**********...**********; $c_5 = 8$
s_6 = **11111111**********...**********; $c_6 = 8$
s_7 = **11111111********; $c_7 = 8$
s_8 = **11111111; $c_8 = 8$
s_{opt}=11111111111111111111111111111111111111111111111111111111111111111

Figure 4.1 An optimal string broken up into eight building blocks. The function $R_1(x)$ (where x is a bit string) is computed by summing the coefficients c_s corresponding to each of the given schemas of which x is an instance. For example, $R_1(1111111100\ldots0) = 8$, and $R_1(1111111100\ldots011111111) = 16$. Here $c_s = order(s)$.

Given the Building Block Hypothesis, one might expect that the building-block structure of R_1 will lay out a "royal road" for the GA to follow to the optimal string. One might also expect that the GA will outperform simple hill-climbing schemes, since a large number of bit positions must be optimized simultaneously in order to move from an instance of a lower-order schema (e.g., $11111111 * * \cdots *$) to an instance of a higher-order intermediate schema (e.g., $11111111 * * * * * * * * 11111111 * * \cdots *$). However, as will be described below, these expectations were overturned.

Experimental Results

We ran a genetic algorithm on R_1 with a population size of 128, and with the initial population generated at random. We used a simple GA with one modification: "sigma truncation" selection was used instead of proportional selection to assign the expected number of offspring to each individual. In our scheme, each individual i's expected number of offspring is

$$1 + (F_i - \bar{F})/2\sigma,$$

where F_i is i's fitness, $\bar{F}$ is the mean fitness of the population, and σ is the standard deviation of the fitnesses in the population. The number of expected offspring of any string was cut off at 1.5—if the above formula gave a higher value, the value was reset to 1.5. This is a strict cutoff, since it implies that most individuals will reproduce only 0, 1, or 2 times. The effect of this selection scheme is to slow down convergence by restricting the effect that a single individual can have on the population, regardless of how much fitter it is than the rest of the population. The single-point crossover rate was 0.7 per pair of parents and the bitwise mutation rate was 0.005.

We compared the GA's performance on R_1 to those of three different iterated hill-climbing methods:

Steepest-ascent hill climbing (SAHC)

1. Choose a string at random. Call this string *current-hilltop*.

2. Going from left to right, systematically flip each bit in the string, one at a time, recording the fitnesses of the resulting one-bit mutants.

3. If any of the resulting one-bit mutants give a fitness increase, then set *current-hilltop* to the one-bit mutant giving the highest fitness increase. (Ties are decided at random.)

4. If there is no fitness increase, then save *current-hilltop* and go to step 1. Otherwise, go to step 2 with the new *current-hilltop*.

5. When a set number of function evaluations has been performed (here, each bit flip in step 2 is followed by a function evaluation), return the highest hilltop that was found.

Next-ascent hill climbing (NAHC)

1. Choose a string at random. Call this string *current-hilltop*.

2. For i from 1 to l (where l is the length of the string), flip bit i; if this results in a fitness increase, keep the new string, otherwise flip bit i back. As soon as a fitness increase is found, set *current-hilltop* to that increased-fitness string without evaluating any more bit flips of the original string. Go to step 2 with the new *current-hilltop*, but continue mutating the new string starting immediately after the bit position at which the previous fitness increase was found.

3. If no increases in fitness were found, save *current-hilltop* and go to step 1.

4. When a set number of function evaluations has been performed, return the highest hilltop that was found.

Random-mutation hill climbing (RMHC)

1. Choose a string at random. Call this string *best-evaluated*.

2. Choose a locus at random to flip. If the flip leads to an equal or higher fitness, then set *best-evaluated* to the resulting string.

3. Go to step 2 until an optimum string has been found or until a maximum number of evaluations have been performed.

4. Return the current value of *best-evaluated*.

(This is similar to a zero-temperature Metropolis method.)

We performed 200 runs of each algorithm, each run starting with a different random-number seed. In each run the algorithm was allowed to continue until the optimum string was discovered, and the total number of function evaluations performed was recorded. The mean and the median number of function evaluations to find the optimum string are

Table 4.1 Mean and median number of function evaluations to find the optimum string over 200 runs of the GA and of various hill-climbing algorithms on R_1. The standard error ($\sigma/\sqrt{\text{number of runs}}$) is given in parentheses.

200 runs	GA	SAHC	NAHC	RMHC
Mean	61,334 (2304)	> 256,000 (0)	> 256,000 (0)	6179 (186)
Median	54,208	> 256,000	> 256,000	5775

given in table 4.1. We compare the mean and the median number of function evaluations to find the optimum string rather than mean and median absolute run time, because in almost all GA applications (e.g., evolving neural-network architectures) the time to perform a function evaluation vastly exceeds the time required to execute other parts of the algorithm. For this reason, we consider all parts of the algorithm other than the function evaluations to take negligible time.

The results of SAHC and NAHC were as expected—whereas the GA found the optimum on R_1 in an average of 61,334 function evaluations, neither SAHC nor NAHC ever found the optimum within the maximum of 256,000 function evaluations. However, RMHC found the optimum on R_1 in an average of 6179 function evaluations—nearly a factor of 10 faster than the GA. This striking difference on landscapes originally designed to be "royal roads" for the GA underscores the need for a rigorous answer to the question posed earlier: "Under what conditions will a genetic algorithm outperform other search algorithms, such as hill climbing?"

Analysis of Random-Mutation Hill Climbing

To begin to answer this question, we analyzed the RMHC algorithm with respect to R_1. (Our analysis is similar to that given for a similar problem on page 210 of Feller 1968.) Suppose the fitness function consists of N adjacent blocks of K ones each (in R_1, $N = 8$ and $K = 8$). What is the expected time (number of function evaluations), $\mathcal{E}(K, N)$, for RMHC to find the optimum string of all ones?

Let $\mathcal{E}(K, 1)$ be the expected time to find a single block of K ones. Once it has been found, the time to discover a second block is longer, since some fraction of the function evaluations will be "wasted" on testing mutations inside the first block. These mutations will never lead to a higher or equal fitness, since once a first block is already set to all ones, any mutation to those bits will decrease the fitness. The proportion of *nonwasted* mutations is $(KN - K)/KN$; this is the proportion of mutations that occur in the $KN - K$ positions outside the first block. The expected time $\mathcal{E}(K, 2)$ to find a second block is

$$\begin{aligned}\mathcal{E}(K, 2) &= \mathcal{E}(K, 1) + \mathcal{E}(K, 1)[KN/(KN - K)] \\ &= \mathcal{E}(K, 1) + \mathcal{E}(K, 1)N/(N - 1).\end{aligned}$$

(If the algorithm spends only $1/m$ of its time in useful mutations, it will require m times as long to accomplish what it could if no mutations were wasted.) Similarly,

$$\mathcal{E}(K,3) = \mathcal{E}(K,2) + \mathcal{E}(K,1)(N/(N-2)),$$

and so on. Continuing in this manner, we derive an expression for the total expected time:

$$\begin{aligned}\mathcal{E}(K,N) &= \mathcal{E}(K,1) + \mathcal{E}(K,1)\frac{N}{N-1} + \cdots + \mathcal{E}(K,1)\frac{N}{N-(N-1)} \\ &= \mathcal{E}(K,1)N\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}\right). \qquad (4.6)\end{aligned}$$

(The actual value is a bit larger, since $\mathcal{E}(K,1)$ is the expected time to the first block, whereas $\mathcal{E}(K,N)$ depends on the worst time for the N blocks (Richard Palmer, personal communication).) By a well-known identity, the right side of equation 4.6 can be written as $\mathcal{E}(K,1)N(\ln N + \gamma)$, where $\ln N$ is the natural logarithm of N and $\gamma \approx 0.5772$ is Euler's constant.

Now we only need to find $\mathcal{E}(K,1)$. A Markov-chain analysis (not given here) yields $\mathcal{E}(K,1)$ slightly larger than 2^K, converging slowly to 2^K from above as $K \to \infty$ (Richard Palmer, personal communication). For example, for $K = 8$, $\mathcal{E}(K,1) = 301.2$. For $K = 8$, $N = 8$, the value of equation 4.6 is 6549. When we ran RMHC on R_1 function 200 times, the average number of function evaluations to the optimum was 6179, which agrees reasonably well with the expected value.

Hitchhiking in the Genetic Algorithm

What caused our GA to perform so badly on R_1 relative to RMHC? One reason was "hitchhiking": once an instance of a higher-order schema is discovered, its high fitness allows the schema to spread quickly in the population, with zeros in other positions in the string hitchhiking along with the ones in the schema's defined positions. This slows the discovery of schemas in the other positions, especially those that are close to the highly fit schema's defined positions. In short, hitchhiking seriously limits the implicit parallelism of the GA by restricting the schemas sampled at certain loci.

The effects of hitchhiking are strikingly illustrated in figure 4.2. The percentage of the population that is an instance of s_i is plotted versus generation for s_1–s_8 for a typical run of the GA on R_1. On this run the schemas s_2, s_4, and s_8 each had two instances in the initial population; none of the other five schemas was present initially. These schemas confer high fitness on their instances, and, as can be seen in figure 4.2, the number of instances grows very quickly. However, the original instances of s_2 and s_4 had a number of zeros in the s_3 loci, and these zeros tended to get passed on to the offspring of instances of s_2 and s_4 along with the desired blocks

of ones. (The most likely positions for hitchhikers are those close to the highly fit schema's defined positions, since they are less likely to be separated from the schema's defined positions under crossover.)

These hitchhikers prevented independent sampling in the s_3 partition; instead, most samples (strings) contained the hitchhikers. As figure 4.2 shows, an instance of s_3 was discovered early in the run and was followed by a modest increase in number of instances. However, zeros hitchhiking on instances of s_2 and s_4 then quickly drowned out the instances of s_3. The very fast increase in strings containing these hitchhikers presumably slowed the rediscovery of s_3; even when it was rediscovered, its instances again were drowned out by the instances of s_2 and s_4 that contained the hitchhikers. The same problem, to a less dramatic degree, is seen for s_1 and s_6. The effectiveness of crossover in combining building blocks is limited by early convergence to the wrong schemas in a number of partitions. This seems to be one of the major reasons for the GA's poor performance on R_1 relative to RMHC.

We observed similar effects in several variations of our original GA. Hitchhiking in GAs (which can cause serious bottlenecks) should not be too surprising: such effects are seen in real population genetics. Hitchhiking in GAs (also called "spurious correlation") has previously been discussed by Schraudolph and Belew (1992), Das and Whitley (1991), and Schaffer, Eshelman, and Offutt (1991), among others.

An Idealized Genetic Algorithm

Why would we ever expect a GA to outperform RMHC on a landscape like R_1? In principle, because of implicit parallelism and crossover. If implicit parallelism works correctly on R_1, then each of the schemas competing in the relevant partitions in R_1 should have a reasonable probability of receiving some samples at each generation—in particular, the schemas with eight adjacent ones in the defining bits should have a reasonable probability of receiving some samples. This amounts to saying that the sampling in each schema region in R_1 has to be reasonably independent of the sampling in other, nonoverlapping schema regions. In our GA this was being prevented by hitchhiking—in the run represented in figure 4.2, the samples in the s_3 region were not independent of those in the s_2 and s_4 regions.

In RMHC the successive strings examined produce far from independent samples in each schema region: each string differs from the previous string in only one bit. However, it is the constant, systematic exploration, bit by bit, never losing what has been found, that gives RMHC the edge over our GA.

Under a GA, if each partition were sampled independently and the best schema in each partition tended to be selected—most likely on differ-

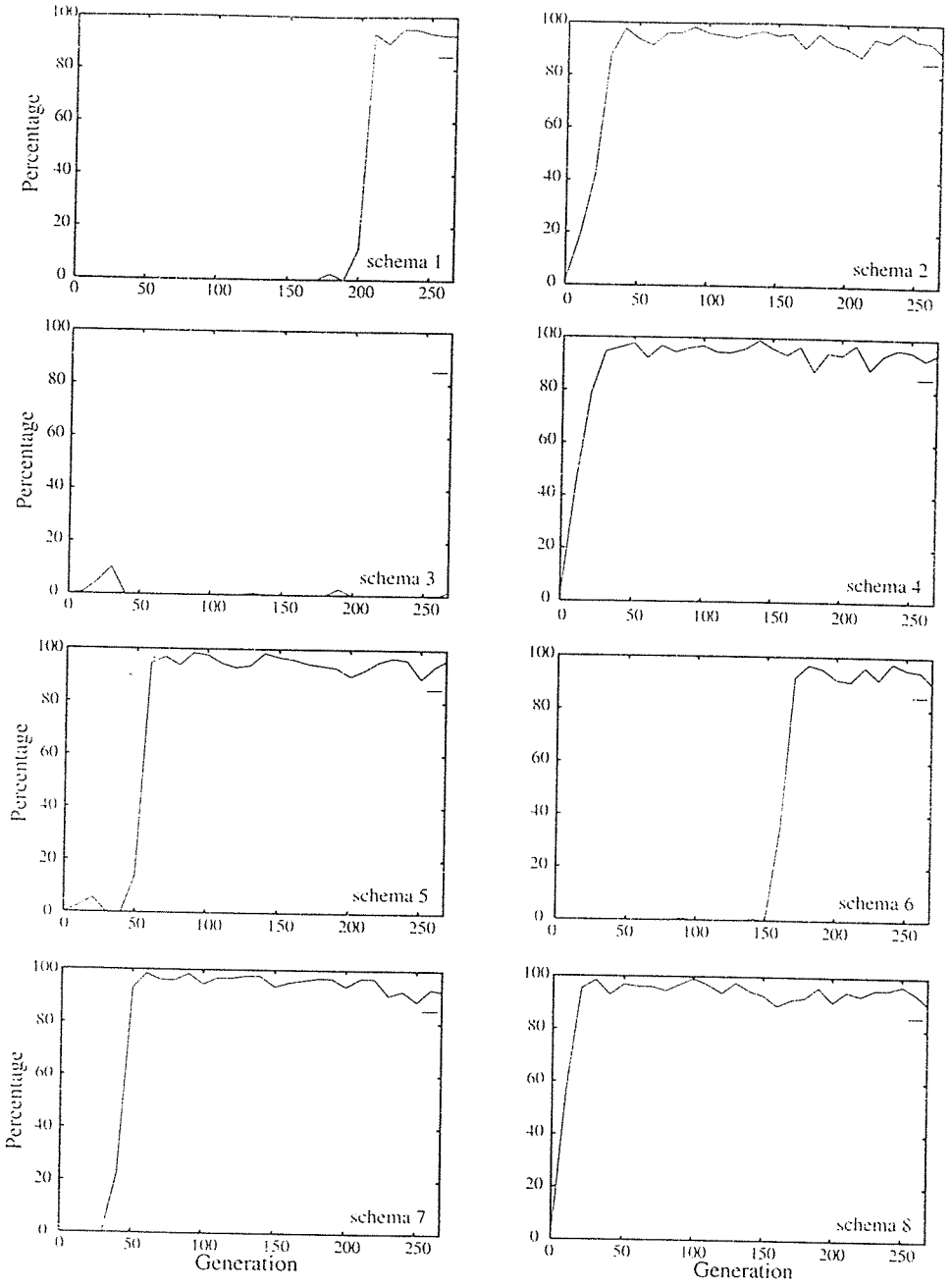

Figure 4.2 Percentage of the population that is an instance of the given schema (1–8) plotted versus generation for a typical GA run on R_1. The data are plotted every 10 generations.

ent strings—then in principle crossover should quickly combine the best schemas in different partitions to be on the same string. This is basically the "Static Building Block Hypothesis" described above. The problems encountered by our GA on R_1 illustrate very clearly the kinds of "biased sampling" problems described by Grefenstette (1991b).

Would an "idealized genetic algorithm" that actually worked according to the SBBH be faster than RMHC? If so, is there any way we could make a real genetic algorithm work more like the idealized genetic algorithm?

To answer this, we defined an idealized genetic algorithm (IGA) as follows (Mitchell, Holland, and Forrest 1994). (Note that there is no population here; the IGA works on one string at a time. Nonetheless, it captures the essential properties of a GA that satisfies the SBBH.)

On each time step, choose a new string at random, with uniform probability for each bit.

The first time a string is found that contains one or more of the desired schemas, sequester that string.

When a string containing one or more not-yet-discovered schemas is found, instantaneously cross over the new string with the sequestered string so that the sequestered string contains all the desired schemas that have been discovered so far.

How does the IGA capture the essentials of a GA that satisfies the SBBH? Since each new string is chosen completely independently, all schemas are sampled independently. Selection is modeled by sequestering strings that contain desired schemas. And crossover is modeled by instantaneous crossover between strings containing desired schemas. The IGA is, of course, unusable in practice, since it requires knowing precisely what the desired schemas are, whereas in general (as in the GA and in RMHC) an algorithm can only measure the fitness of a string and does not know ahead of time what schemas make for good fitness. But analyzing the IGA can give us a lower bound on the time any GA would take to find the optimal string of R_1. Suppose again that our desired schemas consist of N blocks of K ones each. What is the expected time (number of function evaluations) until the sequestered string contains all the desired schemas? (Here one function evaluation corresponds to the choice of one string.) Solutions have been suggested by Greg Huber and by Alex Shevoroskin (personal communications), and a detailed solution has been given by Holland (1993). Here I will sketch Huber's solution.

First consider a *single* desired schema H (i.e., $N = 1$). Let p be the probability of finding H on a random string (here $p = 1/2^K$). Let q be the prob-

ability of not finding H: $q = 1 - p$. Then the probability $\mathcal{P}_1(t)$ that H will be found by time t (that is, at any time step between 0 and t) is

$$\begin{aligned}\mathcal{P}_1(t) &= 1 - \text{Probability that } H \text{ will } \textit{not} \text{ be found by time } t \\ &= 1 - q^t.\end{aligned}$$

Now consider the case with N desired schemas. Let $\mathcal{P}_N(t)$ be the probability that all N schemas have been found by time t:

$$\mathcal{P}_N(t) = (1 - q^t)^N.$$

$\mathcal{P}_N(t)$ gives the probability that all N schemas will be found *sometime* in the interval $[0, t]$. However, we do not want $\mathcal{P}_N(t)$; we want the expected time to find all N schemas. Thus, we need the probability $P_N(t)$ that the last of the N desired schemas will be found at exactly time t. This is equivalent to the probability that the last schema will not be found by time $t - 1$ but will be found by time t:

$$\begin{aligned}P_N(t) &= \mathcal{P}_N(t) - \mathcal{P}_N(t-1) \\ &= (1 - q^t)^N - (1 - q^{t-1})^N.\end{aligned}$$

To get the expected time $\mathcal{E}_N$ from this probability, we sum over t times the probability:

$$\begin{aligned}\mathcal{E}_N &= \sum_{t=1}^{\infty} t \; P_N(t) \\ &= \sum_{t=1}^{\infty} t \; ((1 - q^t)^N - (1 - q^{t-1})^N).\end{aligned}$$

The expression $(1 - q^t)^N - (1 - q^{t-1})^N$ can be expanded in powers of q via the binomial theorem and becomes

$$\left[\binom{N}{1}\left(\frac{1}{q} - 1\right)q^t\right] - \left[\binom{N}{2}\left(\frac{1}{q^2} - 1\right)q^{2t}\right]$$
$$+ \left[\binom{N}{3}\left(\frac{1}{q^3} - 1\right)q^{3t}\right] - \cdots - \left[\binom{N}{N}\left(\frac{1}{q^N} - 1\right)q^{Nt}\right].$$

(N is arbitrarily assumed to be even; hence the minus sign before the last term.)

Now this entire expression must be multiplied by t and summed from 1 to ∞. We can split this infinite sum into the sum of N infinite sums, one from each of the N terms in the expression above. The infinite sum over the first term is

$$
\begin{aligned}
&\binom{N}{1}\left(\frac{1}{q}-1\right)\sum_{t=1}^{\infty} tq^t \\
&= \binom{N}{1}\left(\frac{1}{q}-1\right)\left(q+2q^2+3q^3+\cdots\right) \\
&= \binom{N}{1}\left(\frac{1}{q}-1\right) q\left(1+2q+3q^2+\cdots\right) \\
&= \binom{N}{1}\left(\frac{1}{q}-1\right) q\frac{d}{dq}\left(q+q^2+q^3+\cdots\right) \\
&= \binom{N}{1}\left(\frac{1}{q}-1\right) q\frac{d}{dq}\left(\frac{q}{1-q}\right) \qquad \text{(using a well-known identity for } 0 \le q < 1\text{)} \\
&= \binom{N}{1}\left(\frac{1}{q}-1\right) q\left(\frac{1}{1-q}\right)^2 \\
&= \binom{N}{1}\frac{1}{1-q}.
\end{aligned}
$$

Similarly, the infinite sum over the nth term of the sum can be shown to be

$$\binom{N}{n}\frac{1}{1-q^n}.$$

Recall that $q = 1 - p$, and $p = 1/2^K$. If we substitute $1 - p$ for q and assume that p is small so that $q^n = (1 - p)^n \approx 1 - np$, we obtain the following approximation:

$$\mathcal{E}_N \approx \frac{1}{p}\left[\frac{\binom{N}{1}}{1} - \frac{\binom{N}{2}}{2} + \frac{\binom{N}{3}}{3} - \cdots - \frac{\binom{N}{N}}{N}\right]. \tag{4.7}$$

For $N = 8$, $K = 8$ the approximation gives an expected time of approximately 696, which is the exact result we obtained as the mean over 200 runs of a simulation of the IGA (Mitchell, Holland, and Forrest 1994). (The standard error was 19.7.)

The sum in the brackets in equation 4.7 can be evaluated using the following identity derived from the binomial theorem and from integrating $(1 + x)^N$:

$$\sum_{n=1}^{N}\binom{N}{n}\frac{x^n}{n} = \sum_{n=1}^{N}\frac{1}{n}[(1+x)^n - 1].$$

Let $x = -1$. Then we can simplify equation 4.7 as follows:

$$\mathcal{E}_N \approx -\frac{1}{p}\sum_{n=1}^{N}\binom{N}{n}\frac{(-1)^n}{n}$$

$$= \frac{1}{p}\sum_{n=1}^{N}\frac{1}{n}$$

$$\approx \frac{1}{p}(\ln N + \gamma)$$

$$= 2^K(\ln N + \gamma).$$

Setting aside the details of this analysis, the major point is that the IGA gives an expected time that is on the order of $2^K \ln N$, whereas RMHC gives an expected time that is on the order of $2^K N \ln N$—a factor of N slower. This kind of analysis can help us understand how and when the GA will outperform hill climbing.

What makes the IGA faster than RMHC? To recap, the IGA perfectly implements implicit parallelism: each new string is completely independent of the previous one, so new samples are given independently to each schema region. In contrast, RMHC moves in the space of strings by single-bit mutations from an original string, so each new sample has all but one of the same bits as the previous sample. Thus, each new string gives a new sample to only one schema region. The IGA spends more time than RMHC constructing new samples; however, since we are counting only function evaluations, we ignore the construction time. The IGA "cheats" on each function evaluation, since it knows exactly what the desired schemas are, but in this way it gives a lower bound on the number of function evaluations that the GA will need.

Independent sampling allows for a speedup in the IGA in two ways: it allows for the possibility that multiple schemas will appear simultaneously on a given sample, and it means that there are no wasted samples as there are in RMHC (i.e., mutations in blocks that have already been set correctly). Although the comparison we have made is with RMHC, the IGA will also be significantly faster on R_1 (and similar landscapes) than any hill-climbing method that works by mutating single bits (or a small number of bits) to obtain new samples.

The hitchhiking effects described earlier also result in a loss of independent samples for the GA. The goal is to have the GA, as much as possible, approximate the IGA. Of course, the IGA works because it explicitly knows what the desired schemas are; the GA does not have this information and can only estimate what the desired schemas are by an implicit sampling procedure. But it is possible for the GA to approximate a number of the features of the IGA:

Independent samples The population has to be large enough, the selection process has to be slow enough, and the mutation rate has to be sufficiently high to make sure that no single locus is fixed at a single value in every string in the population, or even in a large majority of strings.

Sequestering desired schemas Selection has to be strong enough to preserve desired schemas that have been discovered, but it also has to be slow enough (or, equivalently, the relative fitness of the nonoverlapping desirable schemas has to be small enough) to prevent significant hitchhiking on some highly fit schemas, which can crowd out desired schemas in other parts of the string.

Instantaneous crossover The crossover rate has to be such that the time for a crossover that combines two desired schemas to occur is small with respect to the discovery time for the desired schemas.

Speedup over RMHC The string has to be long enough to make the factor of N speedup significant.

These mechanisms are not all mutually compatible (e.g., high mutation works against sequestering schemas), and thus they must be carefully balanced against one another. These balances are discussed in Holland 1993, and work on using such analyses to improve the GA is reported in Mitchell, Holland, and Forrest 1994.

4.3 EXACT MATHEMATICAL MODELS OF SIMPLE GENETIC ALGORITHMS

The theory of schemas makes predictions about the expected change in frequencies of schemas from one generation to the next, but it does not directly make predictions concerning the population composition, the speed of population convergence, or the distribution of fitnesses in the population over time. As a first step in obtaining a more detailed understanding of and making more detailed predictions about the behavior of GAs, several researchers have constructed "exact" mathematical models of simple GAs (see, e.g., Goldberg 1987; Goldberg and Segrest 1987; Davis and Principe 1991; Vose and Liepins 1991; Nix and Vose 1991; Horn 1993; Vose 1993; Whitley 1993a). These exact models capture every detail of the simple GA in mathematical operators; thus, once the model is constructed, it is possible to prove theorems about certain interesting properties of these operators. In this section I will sketch the model developed by Vose and Liepins (1991) and summarize extensions made by Nix and Vose (1991) and by Vose (1993).