Stochastic optimization: a review

Dimitris Fouskakis[*] and David Draper[†]

28 November 2001

Abstract

We review three leading stochastic optimization methods—simulated annealing, genetic algorithms, and tabu search. In each case we analyze the method, give the exact algorithm, detail advantages and disadvantages, and summarize the literature on optimal values of the inputs. As a motivating example we describe the solution—using Bayesian decision theory, via maximization of expected utility—of a variable selection problem in generalized linear models, which arises in the cost-effective construction of a patient sickness-at-admission scale as part of an effort to measure quality of hospital care.

Keywords: Bayesian decision theory, genetic algorithms, heuristic methods, hybrid algorithms, local search, maximization of expected utility, simulated annealing, variable selection, tabu search.

1 Introduction

In the past 50 years, since the development of digital computers, many investigators have studied the problem of numerically optimizing an objective function. One approach is *stochastic optimization*, in which the search for the optimal solution involves randomness in some constructive way. If $\mathcal{S}$ denotes the (finite) set of all possible solutions, the task we consider is to maximize or minimize the *objective function* $f\colon \mathcal{S} \to \mathbb{R}$. In the case of maximization, on which we focus here, the problem is to find a *configuration* $x_{opt} \in \mathcal{S}$ which satisfies

$$f(x_{opt}) \geq f(x) \quad \text{for all } x \in \mathcal{S}. \tag{1}$$

All of the optimization methods we consider have the character of a discrete-time search chain, in which an initial member x_0 of $\mathcal{S}$ is chosen by some means and becomes the *current configuration* x_t at time $t = 0$, and the algorithm then iteratively repeats the process of deciding on a *move* from x_t to a *proposed configuration* x_{t+1}. Many of the algorithms we examine rely on a *neighborhood* structure in deciding where to move next; this requires a rule, often based on a measure of distance, which uniquely identifies all of the neighbors of a given current configuration.

It is easy to see that as the dimension of $\mathcal{S}$ increases the harder the task becomes, and more time is needed to find the optimal, or at least a near-optimal, configuration. Another difficulty in this problem is that it is common for the objective function to have many *local optima*. An algorithm like the well-known *local search* (LS; e.g., Aarts and Korst 1989; Algorithm 1), which only accepts moves with higher values of the objective function than the previous move, will not perform well in this situation, since it is likely that the search will get stuck in a local optimum.

The disadvantages of LS algorithms can be formulated as follows:

- By definition, such algorithms terminate in a local maximum, and there is generally no information as to the amount by which this local maximum falls short of a global maximum;
- The obtained local maximum depends on the initial configuration, for the choice of which generally no guidelines are available; and

[*]Department of Mathematical Sciences, University of Bath, Claverton Down, Bath BA2 7AY (email `df@maths.bath.ac.uk`).

[†]Department of Applied Mathematics and Statistics, Baskin School of Engineering, University of California, 1156 High Street, Santa Cruz CA 95064, USA (email `draper@ams.ucsc.edu`, web `http://www.soe.ucsc.edu/~draper`).

Algorithm 1: Local Search (LS)

```
Begin;
  Choose a random configuration $x_0$;
  Set $x := x_0$;
  Repeat:
    Generate a new configuration $y$ from the neighborhood of $x$;
    If $f(y) \geq f(x)$ then $x := y$;
  Until $f(y) < f(x)$ for all $y$ in the neighborhood of $x$;
End.
```

- It is typically not possible to give an upper bound for the computation time.

To avoid some of these disadvantages, a number of potential improvements are possible:

- Execution of the algorithm for a large number of initial configurations, say M, at the cost of an increase in computation time; for $M \to \infty$, in the case in which the number of elements of $\mathcal{S}$ is finite, such an algorithm finds a global maximum with probability 1, if only because a global maximum is encountered as an initial configuration with probability 1 as $M \to \infty$;

- Use of information gained from previous runs of the algorithm to improve the choice of an initial configuration for the next run;

- Introduction of a more complex move-generation mechanism, in order to be able to "jump away from" the local maxima corresponding to the simple approach to generating moves. To choose this more complex move-generation mechanism properly requires detailed knowledge of the problem itself; and

- Acceptance of moves which correspond to a decrease in the objective function in a limited way, in the hope that this will lead to a higher local maximum.

In this paper we review three leading stochastic optimization methods (plus several variations on them)—*simulated annealing* (SA), *genetic algorithms* (GA), and *tabu search* (TS)—each of which (because of its use of ideas like the ones above) often manages to avoid the disadvantages of LS algorithms. In Section 2 below we give details of a complicated optimization problem which serves to illustrate the central features of each of these three methods. Sections 3–5 are devoted to the three algorithms in turn; in each case we analyze the method, give the exact algorithm, detail advantages and disadvantages, and summarize the literature on optimal values of the inputs. In Section 6 we present some results on the performance of SA, GA, and TS in the optimization problem described in Section 2, and Section 7 concludes with a comparative discussion.

2 An example: Bayesian variable selection via maximization of expected utility

An important topic in the field of health policy is the assessment of the quality of health care offered to hospitalized patients (e.g, Jencks et al. 1988, Kahn et al. 1988, Draper et al. 1990). One way to make such an assessment that has gained considerable attention in countries including the US and UK over the past 15 years (e.g., Daley et al. 1988, Draper 1995, Goldstein and Spiegelhalter 1996) is *league-table* or *input-output* (IO) quality assessment, in which outputs of a number of hospitals—such as patient mortality—are compared after adjusting for inputs. The principal relevant input is patient sickness at admission, and—if this approach is to be applied at a large number of hospitals—it is important to devise a cost-effective method of measuring admission sickness. We focus here on mortality within 30 days of admission, which is the standard outcome used in IO quality assessment for hospitalized patients.

The traditional way to construct an admission sickness scale (e.g., Keeler et al. 1990) is (a) to gather a large number of sickness indicators on a representative sample of patients with a given disease (we focus here on pneumonia), together with their 30-day death outcomes, and (b) to fit a variety of logistic regression models (e.g., Hosmer and Lemeshow 1989) predicting death status from the sickness variables, using standard frequentist variable-selection methods such as all-subsets regression (e.g., Weisberg 1985) to choose a parsimonious subset of the available predictors on which to base the scale. In a major US study (Kahn, Rubenstein et al. 1990)—conducted by the Rand Corporation—of quality of hospital care for elderly patients in the late 1980s, this approach was used to select a core of 14 predictors from the list of $p = 83$ available sickness indicators for pneumonia. When input-output quality assessment is to be conducted under budgetary constraints, however, as will almost always be the case, this method—which might be termed a *benefit-only* approach to variable selection—is sub-optimal, because it takes no account of the *costs* of data collection of the available sickness predictors, which may vary considerably in practice (for pneumonia in the Rand study the range in costs, as measured by the time needed to abstract each variable, was from 30 seconds to 10 minutes, a factor of 20 to 1). What is needed is a *cost-benefit* tradeoff (Draper 1996), in which variables that cost too much given how well they predict death are discarded. Draper and Fouskakis (2000) and Fouskakis (2001) detail one approach to this cost-benefit tradeoff using Bayesian decision theory, as follows.

Suppose (a) the 30–day mortality outcome y_i and data on p sickness indicators $(s_{i1}, \dots, s_{ip})$ have been collected on n individuals sampled randomly from a population $\mathcal{P}$ of patients with a given disease, and (b) the goal is to predict the death outcome for n^* new patients who will in the future be sampled randomly from $\mathcal{P}$, (c) on the basis of some or all of the predictors s_j, when (d) the marginal costs of data collection per patient $c_1, \dots, c_p$ for the s_j vary considerably. What is the best subset of the s_j to choose, if a fixed amount of money is available for this task and you are rewarded based on the quality of your predictions? To solve this problem we maximize expected utility, defined in a way that trades off predictive accuracy against data collection costs. The data on which we illustrate this method here consist of a representative sample of $n = 2,532$ elderly American patients hospitalized in the period 1980–86 with pneumonia, taken from the Rand study mentioned above. Since data on future patients are not available, we use a *cross-validation* approach (e.g., Gelfand et al. 1992, Hadorn et al. 1992) in which (i) a random subset of n_M observations is drawn for creation of the mortality predictions (the *modeling* subsample) and (ii) the quality of those predictions is assessed on the remaining $n_V = (n - n_M)$ observations (the *validation* subsample, which serves as a proxy for future patients).

In the approach presented here utility is quantified in monetary terms, so that the data collection utility is simply the negative of the total amount of money required to gather data on the specified predictor subset. Letting $I_j = 1$ if s_j is included in a given model (and 0 otherwise), the data-collection utility associated with subset $I = (I_1, \dots, I_p)$ for patients in the validation subsample is

$$U_D(I) = -n_V \sum_{j=1}^{p} c_j I_j, \tag{2}$$

where c_j is the marginal cost per patient of data abstraction for variable j (these costs were obtained by approximating the average amount of time needed by qualified nurses to abstract each variable from

Table 1: *Cross-tabulation of actual versus predicted death status. The left-hand table records the monetary rewards and penalties for correct and incorrect predictions; the right-hand table summarizes the frequencies in the* 2×2 *tabulation.*

		Rewards and Penalties		Counts	
		Predicted		Predicted	
		Died	Lived	Died	Lived
Actual	Died	C_{11}	C_{12}	n_{11}	n_{12}
	Lived	C_{21}	C_{22}	n_{21}	n_{22}

medical records and multiplying these times by the mean wage (about US$20 per hour in 1990) for the abstraction personnel).

To measure the accuracy of a model's predictions, a metric is needed which quantifies the discrepancy between the actual and predicted values, and in this problem the metric must come out in monetary terms on a scale comparable to that employed with the data-collection utility. In the setting of this example the actual values y_i are binary death indicators and the predicted values $\hat{p}_i$, based on statistical modeling, take the form of estimated death probabilities. We use an approach to the comparison of actual and predicted values that involves dichotomizing the $\hat{p}_i$ with respect to a cutoff, to mimic the decision-making reality that actions taken on the basis of input-output quality assessment will have an all-or-nothing character at the hospital level (for example, regulators must decide either to subject or not subject a given hospital to a more detailed, more expensive quality audit based on *process* criteria; see, e.g., Kahn, Rogers et al. 1990). Other, continuous, approaches to the quantification of predictive utility are possible (e.g., a log scoring method as in Bernardo and Smith 1994; Lindley 1968 uses squared-error loss to measure predictive accuracy in a less problem-specific framework than the one presented here). See Brown et al. (1999) for an application of decision theory to variable selection in multivariate regression.

In the first step of our approach, given a particular predictor subset I, we fit a logistic regression model to the modeling subsample M and apply this model to the validation subsample V to create predicted death probabilities $\hat{p}_i^I$. In more detail, letting $y_i = 1$ if patient i dies and 0 otherwise, and taking $s_{i1}, \ldots, s_{ik}$ to be the k sickness predictors for this patient under model I, the statistical assumptions underlying logistic regression in this case are

$$\begin{aligned} (y_i \,|\, p_i^I) &\overset{\text{indep}}{\sim} \text{Bernoulli}(p_i^I), \\ \log(\tfrac{p_i^I}{1-p_i^I}) &= \beta_0 + \beta_1 s_{i1} + \ldots + \beta_k s_{ik}. \end{aligned} \tag{3}$$

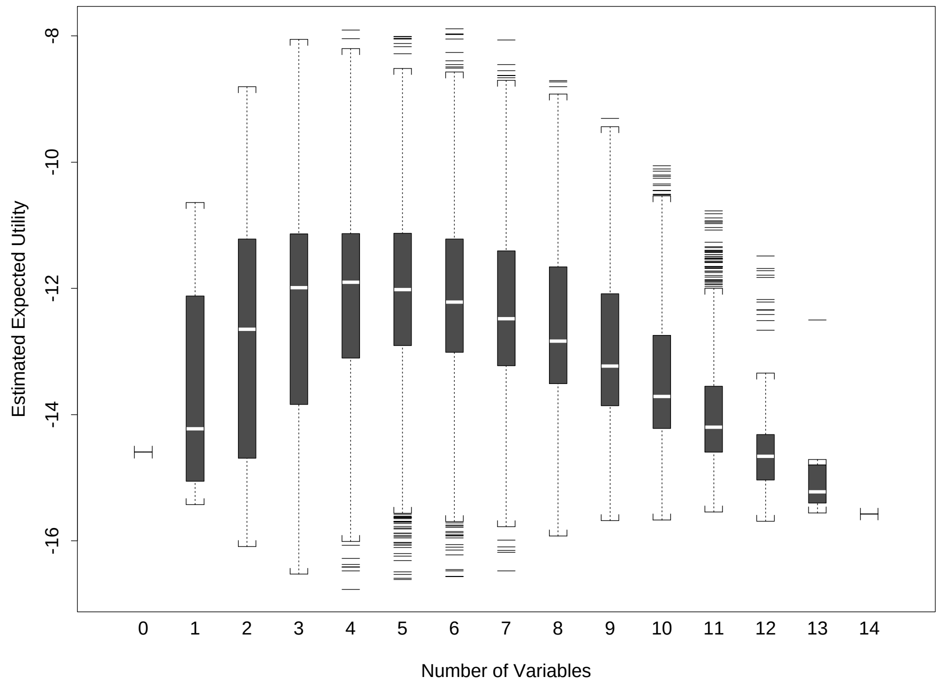
We use maximum likelihood to fit this model (as a computationally efficient approximation to Bayesian fitting with relatively diffuse priors), obtaining a vector $\hat{\beta}$ of estimated logistic regression coefficients, from which the predicted death probabilities for the patients in subsample V are given by

$$\hat{p}_i^I = \left[1 + \exp\left(- \sum_{j=0}^{k} \hat{\beta}_j s_{ij} \right) \right]^{-1}, \tag{4}$$

where $s_{i0} = 1$ ($\hat{p}_i^I$ may be thought of as the sickness score for patient i under model I).

In the second step of our approach we classify patient i in the validation subsample as predicted dead or alive according to whether $\hat{p}_i^I$ exceeds or falls short of a cutoff p^*, which is chosen—by searching on a discrete grid from 0.01 to 0.99 by steps of 0.01—to maximize the predictive accuracy of model I.

Figure 1: *Estimated expected utility as a function of number of predictors retained, with* $p = 14$, $\left(\frac{n_M}{n}, \frac{n_V}{n}\right) = \left(\frac{2}{3}, \frac{1}{3}\right)$, *and* $(C_{11}, C_{12}, C_{21}, C_{22}) = (34.8, -139.2, -69.6, 8.7)$.



We then cross-tabulate actual versus predicted death status in a 2×2 contingency table, rewarding and penalizing model I according to the numbers of patients in the validation sample which fall into the cells of the right-hand part of Table 1. The left-hand part of this table records the rewards and penalties in US\$. The predictive utility of model I is then

$$U_P(I) = \sum_{l=1}^{2} \sum_{m=1}^{2} C_{lm} \, n_{lm}. \tag{5}$$

We use values of the C_{lm} based on expert judgment guided by the decision-making realities of incorrectly conducting or failing to conduct an expensive process audit at a given hospital (see Fouskakis 2001 for details). The overall expected utility function to be maximized over I is then simply

$$E\left[U(I)\right] = E\left[U_D(I) + U_P(I)\right], \tag{6}$$

where this expectation is over all possible cross-validation splits of the data. The number of such splits is far too large to evaluate the expectation directly; in practice we therefore use Monte Carlo methods to evaluate it, averaging over N random modeling and validation splits. When different optimization methods are compared to see which is the best at finding the global maximum of (6), to make the comparison fair they must all be given the same amount of CPU time with which to perform the search, and then the choice of N becomes another optimization variable: if N is small a given method can visit a large number of models but will obtain a noisy estimate of how good those models are, whereas if N is large the estimate of a model's quality will be more precise but the number of models that can be visited given the time constraint will be much smaller.

Figure 1, from Fouskakis (2001), provides an illustration of this methodology in action. Since the best way to compare optimization methods on the quality of the configurations they find is to know the truth in advance, we evaluated (6) using $N = 500$ modeling/validation splits for all $2^{14} = 16,384$ possible subsets of the $p = 14$ variables in the Rand pneumonia sickness scale mentioned above (this choice of N was large enough to yield a Monte Carlo standard error for each expected utility estimate of only about US\$0.05, which is small enough to reliably identify the good models), and Figure 1 presents parallel boxplots of all 16,384 estimated expected utility values as a function of the number k of variables included in the scale (in this problem a configuration is a binary vector of length p). It can be seen, as is intuitively reasonable, that as k increases from 0 the models get better on average up to a maximum or near-maximum at $k = 3$–7 and then get steadily worse up to $k = 14$.

Of course, we have the luxury of knowing the right answer here only because the space of possible configurations was relatively small. Searching through this space with all $p = 83$ available sickness variables, where the total number of possible subsets is $2^{83} \doteq 9.7 \cdot 10^{24}$, is out of the question, motivating the need for global optimization methods such as simulated annealing, to which we now turn.

3 Simulated annealing (SA)

Simulated annealing (SA; Kirkpatrick et al. 1983) is a discrete optimization method that was developed in the early 1980s. Its genesis was a set of ideas first put forward by Metropolis et al. (1953) to simulate a system of particles undergoing a change in temperature. Under perturbation such a system attempts to find an equilibrium point that minimizes the total energy, and Metropolis et al. tried to predict this equilibrium point using statistical thermodynamics (annealing refers to the cooling of materials under controlled conditions in a heat bath). Kirkpatrick et al. noticed that an analogy could be made between system states in the Metropolis problem and possible configurations in a more general optimization problem, with the value of the objective function playing the role of energy and Metropolis' temperature corresponding to a control parameter in the optimization process.

SA is a *stochastic local search* technique which approximates the maximum of the objective function $f\colon \mathcal{S} \to \mathbb{R}$ over a finite set $\mathcal{S}$. It operates iteratively by choosing an element y from a neighborhood $N(x)$ of the present configuration x; the candidate y is either accepted as the new configuration or rejected. SA may accept y with positive probability even if y is worse than x, which permits the algorithm to avoid getting stuck in local maxima by "climbing downhill." SA has proven successful in many applications (e.g., Van Laarhoven and Aarts 1988); a large theoretical and empirical literature can thus be drawn upon for ideas on the implementation of SA in any given problem.

Let $p(x, y, T)$ be the probability of accepting a candidate move to y given the present configuration x. This probability is controlled by the *temperature* T, a choice of terminology made by analogy to the physical cooling process described above. Typically the temperature values are chosen independently of the current value of the objective function as a fixed sequence T_t indexed by time t, the *cooling schedule*. A common choice for $p(x, y, T)$ is the *Metropolis acceptance probability*; for $T > 0$ this is given by

$$p(x, y, T) := \left\{ \begin{array}{ll} 1 & \text{if } f(y) \geq f(x) \\ \exp\left[\frac{f(y)-f(x)}{T}\right] & \text{if } f(y) < f(x) \end{array} \right\}. \tag{7}$$

From the form of (7) it is evident that better moves are always accepted, but it is also possible to accept a move to a worse configuration than the present one with probability $\exp\left[\frac{f(y)-f(x)}{T}\right]$. At high temperatures, the system accepts moves almost randomly, regardless of whether they are uphill or down. As the temperature is lowered, the probability of accepting downhill moves drops and the probability of accepting uphill moves rises. Eventually the system "freezes" in a locally or globally maximum state, and no further moves are accepted. The rate at which T decreases as the number of iterations increases is crucial. We discuss the choice of cooling schedule below.

In SA (Algorithm 2), moves away from the current configuration are chosen according to a *proposal distribution*, such as the uniform distribution on the neighborhood $N(x)$. The algorithm is very general,

```
Algorithm 2: Simulated Annealing (SA)
Begin;
  Choose a configuration $x_0$;
  Select the initial and final temperatures $T_0, T_f > 0$;
  Select the temperature schedule;
  Set $x := x_0$ and $T := T_0$;
  Repeat:
    Repeat:
      Choose a new configuration $y$ from the neighborhood of $x$;
      If $f(y) \geq f(x)$ then $x := y$;
      Else
        Choose a random $u$ uniformly in the range (0,1);
        If $u < \exp\left[\frac{f(y)-f(x)}{T}\right]$ then $x := y$, else $x := x$;
    Until iteration count $= n_{iter}$;
    Decrease $T$ according to the temperature schedule;
  Until stopping criterion = true;
  $x$ is the approximation to the optimal configuration;
End.
```

and a number of decisions must be made in order to implement it for the solution of a particular problem. These can be divided into two categories:

- *generic* choices about inputs to the SA algorithm, such as the cooling schedule, the initial and final temperatures, the stopping criterion, and the parameter n_{iter}; and
- *problem-specific* choices, which mainly have to do with the neighborhood structure.

It is clear from the literature that both types of choices can affect the performance of the algorithm. There has been much research into the theoretical convergence properties of SA (e.g., Aarts and Korst 1989). This work provides pointers as to what factors should be considered in making both generic and problem-specific decisions, but of course these choices depend on the nature of the problem under study.

3.1 Generic and problem-specific decisions

The generic decisions in SA basically concern the cooling schedule. The most commonly used schedule in practice involves *geometric* temperature reduction:

$$T_{new} = T_{old}\,(1 - \epsilon). \tag{8}$$

Small values of ϵ often appear to perform best (e.g., Stander and Silverman 1994); many success stories in the literature take $0.01 \leq \epsilon \leq 0.2$, which leads to rather slow cooling. ϵ can also be defined in terms

Table 2: *Families of temperature schedules for simulated annealing.*

Family	Temperature T_i
Straight	$\frac{T_f - T_0}{M-1}(i-1) + T_0$
Geometric	$T_0 \left(\frac{T_f}{T_0} \right)^{\frac{i-1}{M-1}}$
Reciprocal	$\frac{T_f \, T_0 (M-1)}{(T_f \, M - T_0) + (T_0 - T_f) i}$
Logarithmic	$\frac{T_0 T_f [\log(M+1) - \log 2]}{T_f \log(M+1) - T_0 \log 2 + (T_0 - T_f) \log(i+1)}$

of the upper and lower limits for the temperature parameter—T_0 and T_f, respectively—and the final number of iterations M:

$$\epsilon = 1 - \left(\frac{T_f}{T_0} \right)^{M^{-1}} . \tag{9}$$

Another popular choice, suggested by Lundy and Mees (1986), performs only one iteration per temperature but forces slow cooling by specifying

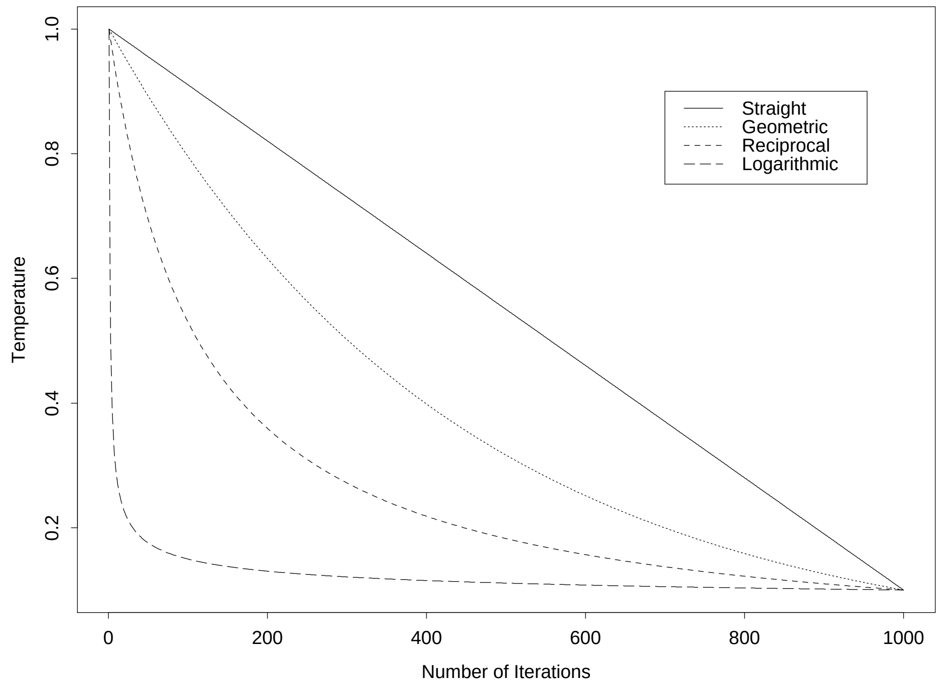
$$T_{new} = \frac{T_{old}}{1 + \beta \, T_{old}}, \tag{10}$$

where β is a suitably small value that can be chosen in a manner analogous to equation (9). The total number of iterations M serves as a stopping criterion once an overall CPU limit is set and an estimate is made of the rate in CPU-seconds at which iterations are made.

A large variety of cooling schedules have been proposed in the literature. In Table 2 we give the most common ones—*straight, geometric, reciprocal*, and *logarithmic*—indexed by the initial and final temperatures T_0 and T_f, the run-length M and the current iteration i, and Figure 2 plots these four schedules with $T_0 = 1.0, T_f = 0.1$, and $M = 1,000$. It appears both from success stories and from theoretical work in the literature that the rate of cooling is more important than the precise schedule used. Thus when SA is used on a new problem a reasonable plan would be to start with a commonly-used schedule such as the geometric and concentrate on fine-tuning the rate parameter (or equivalently the initial and final temperatures), holding the non-geometric schedules in reserve as a further possible source of improvement if satisfactory performance is not immediately attained. $T_0 \doteq 1$ and $0.01 \le T_f \le 0.1$ are popular specifications for initial and final temperatures in the literature (e.g., Fouskakis 2001), although of course the final choices may well need to be problem-specific. Note that there is some redundancy in the combined choice of cooling schedule and (T_0, T_f); for example, the geometric and reciprocal schedules in Figure 2 may be made to look more like the logarithmic curve in that plot by choosing a smaller value of T_f.

Another parameter is the number of iterations at each temperature, n_{iter}, which is often linked to neighborhood size (or even the size of the configuration space $\mathcal{S}$) and may also be a function of temperature (to make sure that a given local maximum has been fully explored it appears to be crucial to ensure that SA spends a lot of time at low temperatures). This can be done by increasing the value of n_{iter} either geometrically (by multiplying by a factor greater than one) or arithmetically (by adding a constant factor) at each new temperature. Also n_{iter} can be determined based on feedback from the process; one strategy that is sometimes used involves accepting a fixed number of moves before the temperature is dropped. The algorithm will typically spend a short amount of time at high temperatures when the acceptance rate is high, but on the other hand it may take an infeasible amount of time to reach the required total number of accepted moves when the temperature is low and the acceptance rate is small.

Figure 2: *The four temperature schedules in Table 2, with* $T_0 = 1.0, T_f = 0.1$, *and* $M = 1,000$.



On the problem-specific choices, as with the generic decisions, it is difficult to identify rules that guarantee good performance, but several themes emerge from the literature. Concerning the neighborhood structure, it is good to be symmetric (i.e., all configurations should have the same number of neighbors), and obviously if y is a neighbor of x, then x should be a neighbor of y as well. To keep the computing time as low as possible it is important for the neighborhood structure to be chosen in such a way that the calculations to be made in every iteration can be carried out quickly and efficiently; thus it is good if the neighborhoods are not large and complex. This has the advantage of rapid neighborhood search but tends to preclude large improvements in a single move. The need to find a middle way between these extremes seems to have been met in the literature mainly by the choice of smaller simpler neighborhoods (Fouskakis 2001).

3.2 Modifications

Here we examine a number of modifications which have proved useful in adapting SA for a variety of different problems. These modifications appear in the literature in only a few examples, so it may be better to consider them only in the case that generic SA fails to provide satisfactory results.

3.2.1 The cooling schedule

If SA is started at an extremely high temperature from a random initial configuration, virtually all moves will be accepted at the beginning, independently of their quality, which means that such moves are no better than the random starting point. One way to avoid this is to cool rapidly at first, for instance by reducing the temperature after a fixed number of acceptances, a strategy that devotes most of the run time to the middle part of the temperature range at which the rate of acceptance is relatively small. Connolly (1990) suggested a constant temperature approach as a different way to solve the high-temperature problem, but the optimal compromise temperature to use throughout the run may well vary not only across problem types but also even within problem type. Dowsland (1993) suggested a potentially useful variation on standard temperature schedules: when a move is accepted, cool the temperature by $T \rightarrow \frac{T}{(1+\beta T)}$, and after each rejection heat it by $T \rightarrow \frac{T}{(1-\gamma T)}$, for $\beta, \gamma > 0$. The choice $\beta = k\,\gamma$ causes SA to make k iterations of heating per iteration of cooling. When the ratio of accepted moves to rejected moves is less than k the system cools down, otherwise it heats. k can be chosen as a function of neighborhood size to ensure that areas close to local optima are searched without a noticeable temperature rise.

3.2.2 The neighborhood structure

SA as described so far uses the assumption that the neighborhood structure is well-defined and unchanging throughout the algorithm, but this need not be true. Improvements may be possible if the neighborhood structure is adjusted as the temperature decreases. One way is to put restrictions on the neighborhood: an example is given by Sechen et al. (1988) in solving the problem of how computer chips should be designed to minimize wasted space on the chip. The task is to optimally place rectangular blocks on the chip area, and these authors permit horizontal and vertical translations of any block to define the neighborhood structure of valid moves. At low temperatures small moves (translations) are the only ones accepted, so to avoid wasting time by proposing and rejecting large moves a limit is created on maximum transition size and this limit is forced to go down as the system cools. Also see Tovey (1988) for an approach in which different neighborhoods are used with probabilities that change as the run unfolds.

3.2.3 The acceptance probability

The use of the Metropolis form (7) for the acceptance probability arises entirely from the laws of thermodynamics; there is no guarantee that this is optimal in any given problem, although theorems about rates of discovery of local optima are available for this schedule (e.g., Aarts and Korst 1989). (7) does at least have the advantage that it accepts downhill moves in such a way that large declines in f have virtually no chance of acceptance, whereas small ones may be accepted regularly.

The biggest problem with Metropolis acceptance probabilities is the algorithm speed. The calculation of $\exp\left[\frac{f(y)-f(x)}{T}\right]$ at every iteration is a time-consuming procedure, and so it might be better to evaluate a cheaper function. Johnson et al. (1989) suggests two possible methods of improvement, one of which is to use the function

$$a(x,y,T) := \left\{ \begin{array}{ll} 1 & \text{if } f(y) \geq f(x) \\ 1 + \frac{f(y)-f(x)}{T} & \text{if } f(y) < f(x) \end{array} \right\}, \tag{11}$$

which approximates the exponential (but note that (11) can go negative for small T). This may be further improved by calculating a look-up table at a series of fixed values over the range of possible values of $\frac{f(y)-f(x)}{T}$. Several other researchers have also found that simpler functions can give good results; for example, Brandimarte et al. (1987) use the form

$$a(x,y,T) := \left\{ \begin{array}{ll} 1 & \text{if } f(y) \geq f(x) \\ \frac{f(x)-f(y)}{T} & \text{if } f(y) < f(x) \end{array} \right\} \tag{12}$$

(but again note that this can exceed 1 for small T). Ogbu and Smith (1990) and Vakharia and Chang (1990) both use probabilities which are independent of $[f(y) - f(x)]$, for problems concerning the optimal sequence in which a given set of tasks should be performed in time.

3.2.4 Parallel implementations: speeding up SA

A major disadvantage of SA is that application of the algorithm may require large amounts of computation time, motivating an interest in accelerating it. In this respect, the increasing availability of *parallel machines* offers an interesting opportunity to explore. Research on parallel implementations of SA has evolved quickly in recent years. The key idea in designing parallel SA algorithms is to efficiently distribute the execution of the various parts of SA over a number of communicating parallel processors.

Aarts and Korst (1989) have considered three approaches to creating parallel versions of SA, of which we consider two. The simplest and most common method is to permit all of the processors to carry out the algorithm with independent sets of random values until it is time to reduce the temperature; then the "master" processor chooses the best configuration found by all of the "slave" processors and all of the slaves begin again from this configuration at the lowered temperature. With this method, when the temperature is high, we expect to have significantly different chains of solutions among the different processors, but when the temperature is low we expect that the processors will end up with solutions very close in terms of neighborhood structure and value of the objective function.

A second method of parallel implementation is to use the processors to visit neighbors at random and independently propose moves. When a move is accepted a signal is sent to the other processors, which turn their attention to the neighborhood of the new current configuration. Again with this strategy almost all proposed configurations will be accepted at high temperatures, so that at this point in the algorithm the potential of parallelism is largely wasted, but at low temperatures most proposals will be rejected, accelerating the search. These two methods can also be combined, by starting with approach one and switching to the second method when the ratio of rejections to acceptances exceeds a specified level.

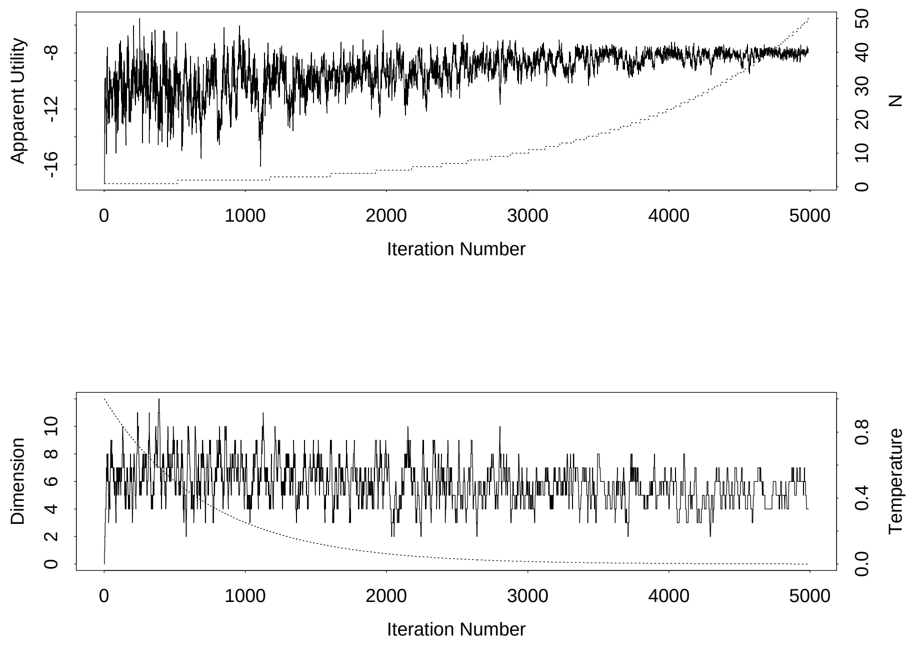
3.2.5 Hybridization

Many researchers have noted that SA can perform better if it is used in combination with other optimization methods. In general this can be done by running these methods before the annealing process is invoked (or after, in order to make an improvement to the configurations encountered by SA). However there are examples of *heuristics* (adaptive, feedback-based search methods) being used as a part of the annealing algorithm. The most common is the use of a pre-processing heuristic to determine a good starting configuration for SA. It is important if this approach is adopted to start the run at a lower temperature than with a random starting value, because otherwise the system will quickly move away from the carefully-chosen initial value in a haphazard way; but if an initial temperature that is too low is used the process may be unable to move away from the good initial configuration. The use of a post-processing heuristic is also common; for example, a local search (Section 1) can be implemented at the end so that the final configuration reported is at least a local maximum.

3.2.6 Sampling

It is standard in SA to sample randomly from the neighborhood of the current configuration. When the search process is near a local optimum, however, most of the neighbors will by definition be worse than where the process is now, so random sampling can lead to acceptance of a downhill move before the local optimum is found. One solution to this problem (e.g., Connolly 1990) is to move through the neighbors cyclically rather than randomly, so that all neighbors have a chance to be chosen before any of them are reconsidered. A second type of problem involving sampling can occur near the end of the SA process: at that point the system is quite cool and a great deal of time can be wasted evaluating potential moves that get rejected out of hand. One way to overcome this (Greene and Supowit 1986)

Figure 3: *Performance of a run of simulated annealing in the Bayesian decision theory problem of Section 2 (reprinted by permission from Draper and Fouskakis 2000). The top panel plots apparent estimated expected utility (left-hand scale) and N (right-hand scale), and the bottom panel displays the model dimension (left-hand scale) and current temperature (right-hand scale), in both cases versus iteration number.*



is to work out the acceptance probability for every possible move in the neighborhood, sample in a weighted fashion using these probabilities as weights, and automatically accept the chosen move.

3.3 Example: Bayesian utility maximization

As an example of SA in action we return to the Bayesian decision theory problem of Section 2, in which the goal was variable selection via maximization of expected utility. As mentioned in that section, expected utility is estimated across N splits of the data into modeling and validation subsets, and (given a fixed budget of CPU time) the choice of N becomes another tuning constant in the search for the global optimum configuration. To explore the tradeoff between visiting many models (small N) and evaluating each model accurately (large N), Figure 3 presents an example of the performance of SA, in a run—based on the $p = 14$ variables in the Rand admission sickness scale for pneumonia—in which SA found the global optimum solution identified in Figure 2. We used a geometric cooling schedule from a starting temperature of 1 to a final temperature of 0.001, and the neighborhood structure was defined by allowing moves from one model to another based on *1–bit flips*: changing a 0 to a 1 or a 1 to a 0 in any single position[1]. Visits to 4,989 models were proposed in the run, with N increasing on an independent geometric schedule from 1 to 50, and the starting value was the null model (with no predictors).

[1]This was accomplished (a) by defining and updating a pointer that scanned cyclically from position 1 to 14 and then back again to 1 (and so on) in the binary string defining the models, and (b) changing a 1 at the current pointer position to 0 or vice versa to create the proposed new model. For example, if the current model is 01100011101001 and the pointer is currently at position 14, the model we consider moving to is 01100011101000; the pointer is then reset to 1; and if this move is accepted the new proposed move is to 11100011101000, otherwise the new proposed move is 11100011101001.

Figure 3 plots four summaries of the run: *apparent* estimated expected utility (based on N modeling/validation splits) and N (the upper panel), and temperature and dimension k of the current model (the lower panel), in all cases as a function of iteration number. During roughly the last half of the run SA mainly examined models with 3–7 predictors (the optimal range in Figure 2), but in the first half SA looked principally at models which, with the hindsight of Figure 2, were far from optimal. In part this is because the early part of the run was based on values of N that were too small: note that in the first 1,000 iterations (when N was at most 2) SA visited a number of models with *apparent* estimated expected utility in excess of -7, whereas no model has *actual* expected utility greater than about -7.9 (Figure 2).

4 Genetic algorithms (GA)

The *genetic algorithm* (GA) was first introduced by Holland (1975), and since then has become a popular method for solving large optimization problems with multiple local optima. Many researchers since have claimed success for GA in a broad spectrum of applications. Holland and his associates at the University of Michigan began to develop GA in the late 1960s, but it was Holland's 1975 book that contained the first full, systematic and theoretical treatment of GA. See Goldberg (1989) and Davis (1991) for a useful description of the algorithm and a number of applications in a range of problems. Interesting applications of the algorithm can also be found in Michalewicz and Janikow (1991), South et al. (1993), Rawlins (1991), Whitley (1992), and Franconi and Jennison (1997). The phrase "genetic algorithm" is more aptly used in the plural, because of the wealth of variations on the basic idea that has grown up since the 1970s; here we will use the abbreviation GA to stand for any of these variations.

4.1 Biological terminology

GA got its name from the process of drawing an analogy between components of the configuration vector x and the genetic structure of a *chromosome*. In this subsection we introduce some of the biological terminology that will appear throughout this section.

Each cell in every living organism contains one or more strings of DNA collected together in chromosomes, which in turn may be divided into one or more *genes*, the parts of the DNA that contain instructions for making proteins. For example, a gene could be responsible for the eye color of the organism. Then the different settings that this eye color can take (e.g., brown, blue, and so on) are the *alleles* of this gene. The position of each gene on the chromosome is that gene's *locus*. The organism may have multiple chromosomes in each cell. The complete collection of genetic material is the organism's *genome*. Two individuals that have identical genomes are said to have the same *genotype*. The organism's *phenotype*—its mental and physical attributes—are determined, under the action of its development before and after birth, by its genotype. Organisms can be *diploid*, when their chromosomes are arrayed in pairs, or *haploid* otherwise. To produce a new off-spring in diploid organisms, a *crossover* operation occurs. In each parent, genes are chosen from the pairs of chromosomes to produce a single chromosome—a *gamete*—and a full collection of chromosomes is then created by pairing up gametes from each parent. The resulting chromosomes are then potentially further modified by *mutation*, in which individual *nucleotides* (the building blocks of DNA) are changed as the parental genetic material is passed on to the offspring. It is convenient to define the *fitness* of the organism either by its probability of living long enough to reproduce or by computing some function of its number of offspring.

In GA the analogy is typically made between a chromosome and a configuration that may be a solution to the optimization problem; most often this is a binary string. The elements of the configuration that play the role of genes are then either single bits or short blocks of adjacent bits, and each allele is a 1 or 0. The crossover operation applied to two given configurations is simply

```
Algorithm 3: Genetic Algorithm (GA)

Begin;
  Randomly generate an even number $n$ of individuals $x^i$ of length $p$;
  Evaluate the fitness $g$ of each individual;
  Repeat:
    Select $n$ new individuals $y^i$ with replacement and probability
      proportional to $g(x^i)$;
    For every pair of $y^i$ do:
      Generate random $r_1$ uniformly in the range $(0, 1)$;
      If $r_1 \le p_c$ then generate a uniform random integer $k$ in the range
        from 1 to $(p - 1)$ inclusive and exchange the $(p - k)$
        elements of each parent $y^i$;
    For every individual $y^i$ do:
      For every binary digit in the individual do:
        Generate random $r_2$ uniformly in the range $(0, 1)$;
        If $r_2 \le p_m$ switch the current element from 0 to 1 or vice versa;
    Save the best $100\gamma\%$ individuals for the next run, with $y_{opt}$ the
      current best;
    Generate the remaining $n(1 - \gamma)$ individuals randomly and
      calculate their fitness $g$;
  Until stopping criterion = true;
  $y_{opt}$ is the approximation to the optimal configuration;
End.
```

an exchange of sections of these possible solutions, while mutation is a random flipping of a bit at a random location along the configuration.

4.2 The basic algorithm

As before the goal is to maximize a function $f(x)$ of the vector $x = (x_1, x_2, \ldots, x_p)$, where here each x_i is binary. (If instead the elements of the configuration vectors are real-valued, one approach is to replace the elements by binary expansions; see Sections 4.4.7–8 below for more ideas in this case.) The basic GA (Algorithm 3) starts by randomly generating an even number n of binary strings of length p to form an initial population:

$$\begin{array}{cccc} x_1^1 & x_2^1 & \ldots & x_p^1 \\ x_1^2 & x_2^2 & \ldots & x_p^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_2^n & \ldots & x_p^n . \end{array} \tag{13}$$

A positive fitness g then is calculated as a monotone increasing function of f for each string in the current generation and n *parents* for the next generation are selected,

$$\begin{array}{cccc} y_1^1 & y_2^1 & \cdots & y_p^1 \\ y_1^2 & y_2^2 & \cdots & y_p^2 \\ \vdots & \vdots & \ddots & \vdots \\ y_1^n & y_2^n & \cdots & y_p^n \end{array} \tag{14}$$

with replacement, with the probability p_j of choosing string j in the current population proportional to its fitness g_j, i.e.,

$$p_j = \frac{g_j}{\sum_{i=1}^n g_i}. \tag{15}$$

The new parents are considered in pairs, and for each pair a crossover operation is performed with a pre-selected probability p_c. If crossover occurs, an integer k is generated uniformly at random between 1 and $(p-1)$ (inclusive) and the last $(p-k)$ elements of each parent are exchanged to create two new strings. For example, for the first pair of parents, suppose that crossover occurred; then two new strings are created,

$$\begin{array}{cccccccc} y_1^1 & y_2^1 & \cdots & y_k^1 & y_{k+1}^2 & y_{k+2}^2 & \cdots & y_p^2 \\ y_1^2 & y_2^2 & \cdots & y_k^2 & y_{k+1}^1 & y_{k+2}^1 & \cdots & y_p^1. \end{array} \tag{16}$$

If crossover does not occur, the parents are copied unaltered into two new strings. After the crossover operation, mutation is performed with a pre-selected probability p_m. If mutation occurs, the value at each string position is switched from 0 to 1 or vice versa, independently at each element of each string. The algorithm is allowed to continue for a specified number of generations. On termination, the string in the final population with the highest value of f can be returned as the solution of the optimization problem. But, since a good solution may be lost during the algorithm, a more efficient strategy is to note the best, or even better the $100\gamma\%$ best, configurations seen at any stage (for some reasonable γ) and return these as the solution. The population size n, parameters p_c and p_m, and fitness function g must be specified before GA is applied. It is often reasonable to take g equal to f, but in some problems a more careful choice may be required (see Sections 4.3 and 4.4.3 below). Note that GA proposes moves away from the configurations currently under examination using a "neighborhood structure" that is completely different from the approach used in SA.

4.3 Implementation of GA

As was the case with SA, the implementer of GA has to make a variety of choices about how to execute the algorithm. For instance, how many times should the algorithm be run, and how should the generations be allowed to evolve? The best number of runs, or of generations, would be a choice large enough to find the optimal solution, but also as small as possible to minimize computation time. One crucial issue in this choice is how many configurations to randomly generate in each repetition of GA; another is how much of the previous population should be retained in the next generation. In one approach the current population is cleared at the end of each repetition and a new initial population is produced randomly at the beginning of the next repetition. By doing this the algorithm is forced to look into new regions, but if the random production of the new population is computationally expensive time is lost. The other extreme is to retain the final population of the previous repetition in the new repetition unchanged; a compromise is to keep a specific percentage ($100\gamma\%$, say) of models from the current population in the new repetition and randomly generate the rest.

Another decision has to do with the population size n, which should depend on the length of the binary strings defining the configurations. Some researchers use small population sizes in order to afford more repetitions of the algorithm (running the risk of under-covering the solution space and failing to find a near-optimal solution), but others prefer to repeat the algorithm fewer times in order to use large population sizes. Neither choice universally dominates in all optimization settings; it appears necessary to spend some effort in the tuning phase of GA, making runs with (small n, large number of generations) and vice versa, to see what works best. Goldberg (1989) claims that the optimal size for binary-coded strings grows exponentially with the length of the string p, but other authors have found that population sizes as small as 30 are often quite adequate in practice even with fairly large p, and Alander (1992) suggests that a value of n between p and $2p$ is not far from optimal for many problems.

Implementation of GA also requires two probabilities, p_c for crossover and p_m for mutation. Here the literature is clearer on the best choices. Almost all researchers agree that the probability of crossover must be fairly high (above 0.3), while the probability of mutation must be quite small (less than 0.1). Many investigators have spent a lot of effort trying to find the best values of these parameters. De Jong (1975), for instance, ran many simulation experiments and concluded that the best population size was 50–100 individuals, the best single-point crossover rate was 0.6 per pair of parents, and the best mutation rate was 0.001 per bit. These settings became widely used in the GA community, even though it was not clear how well GA would perform with these inputs on problems outside De Jong's test suite. Schaffer et al. (1989) spent over a year of CPU time systematically testing a wide range of parameter combinations. They found that the best settings for population size, crossover rate and mutation rate were independent of the problems in their test suite (but improvements to GA since 1989 have since superseded their results). Finally Grefenstette (1986), after much effort, suggested that in small populations (30, for example) it is better to use high values of the crossover rate, such as 0.88. The best crossover rate decreased in his simulations to 0.50 for population size 50 and to 0.30 for population size 80. Finally in most of his runs he used a mutation rate of 0.01. Runs with mutation rate above 0.1 are more like a random search, but values of p_m near 0 were also associated with poor performance.

Finally the GA implementer has to choose a good fitness function g. Theory says that g must be positive and a monotone increasing function of the objective function f. Some researchers prefer their fitness function to take values in the interval $[0, 1]$. The choice is wide; the only guide is that g must provide sufficient selectivity to ensure that the algorithm prefers superior solutions to the extent that it eventually produces an optimal (or at least near-optimal) answer. On the other hand, selectivity must not be so strong that populations polarize at an early stage and the potential advantages of maintaining a diverse collection of solutions are lost. A frequent choice is the simplest—$g = f$—but more complicated fitness functions are sometimes advantageous, e.g., $g = \exp(f)$, or $g = M + f$ for a suitable constant M. If a positive, monotone increasing function of f which has values from 0 to 1 (at least approximately) is desired, a natural choice is

$$g(X) = \frac{f(X) - \min[f(X)]}{\max[f(X)] - \min[f(X)]}, \tag{17}$$

where $\max[f(X)]$ and $\min[f(X)]$ are (at least rough estimators of) the maximum and minimum values of f, respectively.

4.4 Modifications

In this section we examine the most frequent and most effective modifications of the basic GA idea in the literature.

4.4.1 Seeding

What kind of population to use initially—how to *seed* GA—is the subject of some discussion in the literature. The most common idea is to randomly generate strings of 0s and 1s to create the initial population. Some researchers (e.g., Reeves 1992, Kapsalis et al. 1993) believe that starting GA instead with a population of high-quality solutions (obtained, for instance, from another optimization technique) can help the algorithm to find a near-optimal solution more quickly, but this runs the risk of premature convergence to a local optimum that may be far from globally best.

4.4.2 Selection mechanisms

In the original GA, parents are chosen in some random fashion from the population, and then a new set of offspring is generated to replace the parents. One alternative is the *generation gap* of De Jong (1975), in which a proportion R of the population is selected for reproduction and their offspring, when generated, replace an element in the current population at random. From available studies of this question, GA seems to perform best with non-overlapping populations, but with incremental replacement we have the advantage of preventing the occurrence of duplicates, which avoids (a) having to compute the same fitness repeatedly and (b) distorting the process of selection (otherwise duplicate patterns have an increased chance to reproduce).

A second worthwhile alternative is to force the best member(s) of the current population to be member(s) of the next as well. With this method we keep track of the best solution(s) through the whole algorithm. Another useful idea is to compare the parents with the offspring, and instead of copying the offspring directly to the new population, to copy the two best among the four (the two parents and the two children) to the new population, so that (for instance) if the parents are "fitter" than their children, then they both survive. This is called an *elitist* strategy.

Finally, consider the problem of having two chromosomes which are close to different optima and the result of a crossover operator on them is worse than either. To address this Goldberg and Richardson (1987) defined a *sharing function*, such as

$$h(d) = \left\{ \begin{array}{lll} 1 - \frac{d}{D} & \text{if} & d < D \\ 0 & \text{if} & d \geq D \end{array} \right\}, \tag{18}$$

where d is the distance between 2 chromosomes (an obvious measure is *Hamming distance*—a count of the number of discrepant bits—but this may cause problems; for more details refer to Goldberg and Richardson 1987) and D is a parameter. For each pair of chromosomes we evaluate $h(d)$ and calculate

$$\sigma_j = \sum_{i \neq j} h(d_i) \tag{19}$$

for each chromosome j. Then we divide the fitness of each chromosome j by σ_j, and we replace the old fitnesses with the new values. The result of this will be that the fitnesses of close chromosomes will be diminished when compared with those that are far apart.

4.4.3 The fitness calculation

In Section 4.3 we offered several ideas for specifying the fitness function. An alternative approach (e.g., Goldberg 1989) is to use a scaling procedure based on the transformation

$$g = \alpha f + \beta, \tag{20}$$

where f is the objective function and α and β are specified by conditions such as

$$\begin{array}{rcl} \text{mean}(g) & = & \text{mean}(f) \\ \max(g) & = & \mu - \max(f), \end{array} \tag{21}$$

where μ is a constant. A second alternative (e.g., Baker 1985, Whitley 1989, Reeves 1992) is to ignore the objective function altogether and base the selection probabilities on ranks. In this approach children are chosen with probability

$$P([k]) = \frac{2k}{n\,(n+1)}, \tag{22}$$

where $[k]$ is the k^{th} chromosome ranked in ascending order. The best chromosome $[n]$ will be selected with probability $\frac{2}{n+1}$, roughly twice that of the median, whose chance of selection is $\frac{1}{n}$.

Finally a third alternative is to use *tournament selection* (Goldberg and Deb 1991). Suppose we have n chromosomes, and successive blocks of C chromosomes are compared. We choose only the best one as a parent, continuing until the n chromosomes have all been examined, at which point we generate another random permutation. This procedure is repeated until n parents have been chosen, after which each parent is mated with another at random. With this approach the best chromosome will be chosen C times in any series of n tournaments, the worst not at all, and the median on average once.

4.4.4 Crossover operators

In most applications, the simple crossover operator described previously has proved effective, but there are problems where more advanced crossover operators have been found useful. First of all consider the *string-of-chance* crossover. Suppose we have the two chromosomes

$$\begin{array}{c} 1\,1\,0\,1\,0\,0\,0\,1 \\ 1\,1\,0\,1\,1\,1\,0\,0 \end{array} \tag{23}$$

These chromosomes have the same elements in the first four positions. Cases like this are quite common, especially in the later stages of the algorithm. If the crossover point is any of these four first positions, then the new string will not be a different chromosome. To overcome this, the suggestion was made by Booker (1987) and Fairley (1991) to identify crossover points yielding different offspring before undertaking the crossover operation. To implement this Fairley proposed the string-of-chance crossover, which calculates an *exclusive-or (XOR)* between the parents and only permits crossover points at locations between the outermost 1s of the XOR string. In the above example, the result of the XOR operation is

$$0\,0\,0\,0\,1\,1\,0\,1. \tag{24}$$

So only the last four positions will give rise to a different offspring.

In the simple crossover operator we randomly choose a single position and exchange the elements of the two parents, but there is no reason why the choice of crossover points should be restricted to a single position. Many researchers have claimed large improvements with the use of *multi-point crossovers.* Consider the simplest case of *two–point crossover* and suppose we have the following strings:

$$\begin{array}{cc|cccc|cc} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

If the chosen positions are the third and the seventh then we can produce the offspring

$$\begin{array}{c} 0\,1\,0\,0\,0\,1\,0\,0 \\ 1\,0\,1\,0\,1\,1\,1\,0 \end{array} \tag{25}$$

by taking the first two and the last two elements from one parent and the rest from the other each time, but this is of course by no means the only possible exchange; another potentially useful outcome would result from holding the middle block of binary text (positions 3 through 6) constant and exchanging the outer blocks between parents, yielding

$$\begin{array}{c} 1\,0\,1\,0\,1\,1\,1\,0 \\ 0\,1\,0\,0\,0\,1\,0\,0. \end{array} \tag{26}$$

The operator that has received the most attention in recent years is the *uniform crossover.* It was studied in some detail by Ackley (1987) and popularized by Syswerda (1989). Suppose we have the following strings:

$$\begin{gathered}1\,0\,0\,0\,1\,0\,1\,0\\0\,1\,0\,1\,0\,0\,1\,1.\end{gathered} \tag{27}$$

Then at each position randomly (with probability 0.5, say) pick each bit from either of the two parent strings, repeating if it is desired to produce two offspring. One possible outcome is

$$\begin{gathered}0\,0\,0\,1\,0\,0\,1\,0\\1\,1\,0\,0\,1\,0\,1\,1.\end{gathered} \tag{28}$$

In the first offspring, we have chosen the second, third, sixth, seventh, and eighth element from the first parent, and the rest from the second, and in the second offspring we have chosen the first, third, fourth, fifth, sixth and seventh bit from the first parent, and the rest from the second.

A modified version of the uniform crossover is the method used by the *CHC Adaptive Search Algorithm* (Eshelman 1991), which will here be called *highly uniform crossover.* This version crosses over half (or the nearest integer to $\frac{1}{2}$) of the non-matching alleles, where the bits to be exchanged are chosen at random without replacement. For example, if we again have as parents the strings

$$\begin{gathered}1\,0\,0\,0\,1\,0\,1\,0\\0\,1\,0\,1\,0\,0\,1\,1,\end{gathered} \tag{29}$$

there are five non-matching alleles (the first, second, fourth, fifth, and eighth); under highly uniform crossover the recommendation would be to cross three of them randomly, (say) the first, second and fourth, with

$$\begin{gathered}0\,1\,0\,1\,1\,0\,1\,0\\1\,0\,0\,0\,0\,0\,1\,1\end{gathered} \tag{30}$$

as the resulting children. With this operator we always guarantee that the offspring are the maximum Hamming distance from their two parents.

Sirag and Weisser (1987) proposed a different kind of *generalized crossover*, which uses ideas from simulated annealing. In this approach the crossover process is based on a *threshold energy* θ_c which determines the choice of individual bits. In creating the child chromosome a bias toward taking bit $(i+1)$ from the same parent as bit i is introduced, by choosing bit $(i+1)$ from the other parent with probability

$$\exp\left(\frac{-\theta_c}{T}\right), \tag{31}$$

where T is a temperature parameter that is decreased slowly via an annealing schedule. When T is high this behaves like uniform crossover; as T drops the number of switches between parents decreases, mimicking simple crossover; and for low T one of the parents is simply copied. Experimentation is needed as usual with SA to choose θ_c, the initial temperature, and the cooling schedule.

4.4.5 Inversion

Inversion is an operator which takes account of order relationships. It can be used together with the crossover operator to produce a larger variety of offspring. Consider, for instance, the following chromosome:

$$1\,0\,1\,1\,0\,0\,1\,0. \tag{32}$$

We randomly choose two positions with different elements (the second and seventh, say), and exchange these two positions (a *two-bit swap*), yielding

$$1\ 1\ 1\ 1\ 0\ 0\ 0\ 0. \tag{33}$$

Two-bit swaps can also be used in SA to enrich the neighborhood structure (for instance, by first scanning along the bit string using one-bit flips and following this by a round of two-bit swaps), but this use of inversion is different from that in GA.

4.4.6 Adaptive operator probabilities

In simple GA the probability of using either crossover or mutation is fixed throughout, and usually crossover is applied with a high probability (above 30–50%) while mutation is applied with low frequency (less than 1%). Reeves (1992) found it useful to change this approach, and allow the mutation rate to change during the run. To diminish the chance of premature convergence to a local sub-optimum he suggested making the mutation rate inversely proportional to the population diversity. By contrast Booker (1987) used a characteristic called *percent involvement* to create an adaptive crossover rate.

Davis (1991) has further proposed that one or the other of crossover or mutation should be applied at each iteration but not both. With this approach at each step GA chooses which operator to use based on a probability distribution called *operator fitness.* The user can select a value like 75% for crossover fitness, so that crossover would be chosen three times as often as mutation. The idea can be extended by allowing operator fitness to change during the run. One might begin with a high value of crossover fitness, because crossover's importance is higher at the beginning of the run when the population is diverse, and then as diversity decreases the importance of mutation increases as the main way to propose completely new configurations.

4.4.7 Chromosome coding

GA has its most obvious application with binary configurations, but it may also be applied in problems with real-valued inputs. One approach is to simply map the real numbers onto binary strings of desired precision. Thus if an input takes values in the interval $[a, b]$ and is mapped to a string of length p, then the precision is

$$\frac{a-b}{2^p-1}. \tag{34}$$

The problem in this case is that values which are close in the original space may be far away in the new binary-mapped space. For instance, suppose that the optimal real-valued input string is 8, which would be coded (1 0 0 0) with 4-bit chromosomes. The input string 7, which is nearly optimal, comes out (0 1 1 1) in this coding, as far away in Hamming distance from its real-valued neighbor as possible. To overcome this problem Caruna and Schaffer (1988) proposed a gray-scale mapping, but the lack of a simple method for decoding a gray scale introduces further difficulties with their approach.

Examples of comparisons between binary coding and real-valued or even multiple-character coding include the Kitano (1990) many-character representation for graph-generation grammars, the Meyer and Packard (1992) real-valued representation for condition sets, the Montana and Davies (1989) real-valued representation for neural-network weights, and the Schultz–Kremer (1992) real-valued representation for torsion angles in proteins.

4.4.8 Sequence representation

Many situations of interest, such as the *traveling salesman problem* (e.g., Aarts and Korst 1989), can be most naturally represented through *permutations* of the integers from 1 to k. The drawback for GA in such cases is that the original crossover operator does not work. As an example, suppose we have the following two parents:

$$\begin{array}{c} 1\ 5\ 2\ 4\ 3\ 6 \\ 2\ 5\ 3\ 6\ 4\ 1. \end{array} \tag{35}$$

Then if we take the crossover point to be the fourth position we end up with the offspring

$$\begin{array}{c} 1\ 5\ 2\ 6\ 4\ 1 \\ 2\ 5\ 3\ 4\ 3\ 6, \end{array} \tag{36}$$

which are of course invalid.

Several researchers have tried to solve this problem by defining new operators. Goldberg and Lingle (1985) proposed the *partially mapped crossover*. This operator uses two crossover points, and the section between these points defines an *interchange mapping*. In the above example suppose that the two crossover points are the second and fifth:

$$\begin{array}{c|ccc|cc} 1 & 5 & 2 & 4 & 3 & 6 \\ 2 & 5 & 3 & 6 & 4 & 1 \end{array}$$

These crossover points define the interchange mapping

$$\begin{array}{c} 5 \leftrightarrow 5 \\ 2 \leftrightarrow 3 \\ 4 \leftrightarrow 6 \end{array} \tag{37}$$

leading to the offspring

$$\begin{array}{c} 1\ 5\ 3\ 6\ 2\ 4 \\ 3\ 5\ 2\ 4\ 6\ 1 \end{array} \tag{38}$$

via the following rules: (i) the integers between the crossover points are interchanged between the parents, and (ii) the remaining integers in each parent are ranked and replaced by the unused integers in the same rank order.

Reeves (1992) used a *C1 operator* to solve a *flowshop sequencing problem*. This operator randomly chooses a point, takes the first part from one parent and completes the chromosome by taking each legitimate element from the other parent in order. In the example above if we randomly choose the third position

$$\begin{array}{cc|cccc} 1 & 5 & 2 & 4 & 3 & 6 \\ 2 & 5 & 3 & 6 & 4 & 1 \end{array}$$

then we (i) create the "temporary offspring" 1 5 2 5 3 6 4 1 (by taking the 1 5 from the first parent and the rest from the second) and 2 5 1 5 2 4 3 6, and (ii) eliminate duplicates reading from left to right, yielding as the new offspring under the C1 operator

$$\begin{array}{c} 1\ 5\ 2\ 3\ 6\ 4 \\ 2\ 5\ 1\ 4\ 3\ 6. \end{array} \tag{39}$$

Finally, as was the case with binary inputs, a uniform crossover is also useful with permutation problems. We randomly generate a *crossover template* of 0s and 1s, where 1s define elements taken from one parent, while the other elements are copied from the other parent in the order in which they appear in that chromosome. Returning to the example in (35) above, if we generate the template

$$1\ 0\ 1\ 1\ 0\ 0 \tag{40}$$

then we again (i) create the "temporary offspring" 1 ? 2 4 ? ? 2 5 3 6 4 1 (by selecting positions 1, 3, and 4 from parent 1 and the rest from parent 2) and 2 ? 3 6 ? ? 1 5 2 4 3 6, and (ii) use the non-duplicate values in positions 7–12 to fill in the question marks (reading from left to right), yielding as the two new offspring under uniform crossover

$$\begin{array}{c} 1\ 5\ 2\ 4\ 3\ 6 \\ 2\ 1\ 3\ 6\ 5\ 4. \end{array} \tag{41}$$

Mutation also needs to be reconsidered with permutation problems. Reeves (1992) proposed an *exchange mutation*, in which two randomly chosen elements are interchanged (a two-bit swap). Another idea is the *shift mutation*, in which a randomly chosen element is moved a random number of places to the left or right. Finally Davis (1991) used the idea of a *scramble sublist mutation*, in which two points on the string are chosen at random and the elements between these positions are randomly scrambled.

4.4.9 Parallel implementations: speeding up GA

A serious drawback of GA is its inefficiency when implemented on a sequential machine. However, due to its inherent parallel properties, it can be successfully executed on parallel machines, in some cases resulting in a considerable acceleration. One simple idea (e.g., Talbi and Bessière 1991; Mühlenbein et al. 1988) for parallelizing GA is to calculate the fitness of all chromosomes in the current population in parallel. A second approach divides the current population at random into k subpopulations and lets each processor run GA independently on parallel subpopulations for a specified number of generations, perhaps seeding each subpopulation with the best configuration found so far. Pettey et al. (1987) use a version of this idea in which the best configuration is passed along and harvested once per generation; Cohoon et al. (1987) instead wait a fairly large number of generations and then copy randomly chosen subsets of configurations between the subpopulations.

4.4.10 Hybridization

One possible approach to enhancing the effectiveness of GA is to hybridize it with another optimization method. For example, many researchers have described ways in which local neighborhood search or extensions such as SA can be embedded in GA in order to make it more effective.

Goldberg (1989) described *G-bit improvement*, a method for incorporating neighborhood search into a GA (also see Suh and Van Gucht 1987 for a similar hybrid). In this method promising patterns periodically serve as the basis of a neighborhood search defined by 1–bit flips. On completion of the neighborhood search, locally optimal configurations found in this way are re-introduced into the population for the next phase of GA. Kapsalis et al. (1993) and Beaty (1991) divided the search for an optimal configuration into two parts—"top-level" decisions about the form of a solution (made by GA) and problem-specific search methods which refine the GA output.

Two of the best-known hybrids are *genetic local search* (GLS; Algorithm 4) and *genetic simulated annealing* (GSA, e.g., Murata et al. 1996; this is identical to GLS except that SA replaces LS in Algorithm 4). The first of these has been proposed by several authors, mainly for solving the traveling salesman problem (e.g, Jog et al. 1989, Ulder et al. 1991). The problem with both of these algorithms is that it can take a long time to find the locally optimal configurations defining each generation. One way to decrease this computation time can be to search only a part of each neighborhood examined. In GLS, for instance, one can search at random through only a small portion (5–20%, say) of each

Algorithm 4: Genetic Local Search (GLS)

```
Begin;
  (Initialization);
  (Local Search and termination test). Apply Local Search (LS; Algorithm
    1) to each of the $n$ configurations in the current population. If the overall
    stopping criterion (e.g., total CPU time) is met during any of these searches,
    stop. Otherwise let the current population consist of the best configurations
    from each local search and continue;
  (Selection);
  (Crossover);
  (Mutation);
  Keep the best $100\gamma\%$ individuals and randomly generate the remaining
    $n(1-\gamma)$ individuals to create the new population;
  Return to the second step.
```

neighborhood during each of the LS phases, with a given local search concluding if nothing is found that improves on the current local optimum. In GSA it appears to be best (e.g., Murata et al. 1996) to apply SA with constant moderate temperature to each of the n solutions in the current population to avoid losing the value of the current configuration by annealing at high temperatures. To improve the performance of GSA Ishibuchi et al. (1995) suggested a way to modify SA, by selecting a small subset of the current neighborhood and choosing the best configuration in this subset as the candidate for the next transition in SA.

4.5 The genitor algorithm

The *genitor algorithm* (Whitley 1989) has been referred to by Syswerda (1989) as the first of the "steady-state" genetic algorithms. It differs from the original (or "vanilla") version of GA in three ways: during reproduction only one offspring is created; this offspring directly re-enters the population, displacing the currently least fit member (thus after each cycle through the population it will consist of half of the parents (the fittest ones) and all the offspring); and fitness is evaluated by rank (see section 4.4.3), to maintain a more constant selective pressure over the course of the search.

4.6 The CHC adaptive search algorithm

The *CHC adaptive search algorithm* was developed by Eshelman (1991). CHC stands for *cross-generational elitist selection*, *heterogeneous recombination* and *cataclysmic mutation.* This algorithm uses a modified version of uniform crossover, called *HUX*, where exactly half of the different bits of the two parents are swapped. Then with a population size of n, the best n unique individuals from the parent and offspring populations are drawn to create the next generation. *HUX* is the only operator used by CHC adaptive search; there is no mutation.

In CHC adaptive search two parents are allowed to mate only if they are a specified Hamming distance (say d) away from each other, an attempt to encourage diversity by "preventing incest." The usual initial choice is $d = \frac{p}{4}$, where p is the length of the string. If the new population is exactly the same as the previous one, d is decreased and the algorithm is rerun. If d becomes negative this invokes a *divergence procedure*, in which the current population is replaced with n copies of the best member of the previous population, and for all but one member of the current population $r \times p$ bits are flipped at random, where r is the divergence rate (a reasonable compromise value for r is 0.5). d is then replaced by $d = r(1 - r)p$ and the algorithm is restarted.

5 Tabu search (TS)

Tabu search (TS) is a "higher-level" heuristic procedure for solving optimization problems, designed (possibly in combination with other methods) to escape the trap of local optima. Originally proposed by Glover (1977) as an optimization tool applicable to nonlinear covering problems, its present form was proposed 9 years later by the same author (Glover 1986), and with even more details several years later again by Glover (1989). The basic ideas of TS have also been sketched by Hansen (1986). Together with SA and GA, TS was singled out by the *Committee on the Next Decade of Operations Research* (1988) as "extremely promising" for future treatment of practical applications. This stochastic optimization method is well established in the optimization literature but does not appear to be as well known to statisticians (Fouskakis and Draper 1999).

The two key papers on TS are probably Glover (1989, 1990a); the first analytically describes the basic ideas and concerns of the algorithm and the second covers more advanced considerations. Glover (1990b), a tutorial, and Glover et al. (1993), a users' guide to TS, are also helpful. Other authors who have made contributions include Cvijovic and Klinowski (1995), who specialized the algorithm for solving the multiple minima problem for continuous functions, and Reeves (1995).

In a variety of problem settings, TS has found solutions equal or superior to the best previously obtained by alternative methods. A partial list of TS applications is as follows: employee scheduling (e.g., Glover and McMillan 1986); maximum satisfiability problems (e.g., Hansen and Jaumard 1990); telecommunications path assignment (e.g., Oliveira and Stroud 1989); probabilistic logic problems (e.g., Jaumard et al. 1991); neural network pattern recognition (e.g., de Werra and Hertz 1989); machine scheduling (e.g., Laguna, Barnes, et al. 1991); quadratic assignment problems (e.g., Skorin–Kapov 1990); traveling salesman problems (e.g., Malek et al. 1989); graph coloring (e.g., Hertz and de Werra 1987); flow shop problems (e.g., Taillard 1990); job shop problems with tooling constraints (e.g., Widmer 1991); just-in-time scheduling (e.g., Laguna and Gonzalez-Velarde 1991); electronic circuit design (e.g., Bland and Dawson 1991); and nonconvex optimization problems (e.g., Beyer and Ogier 1991).

5.1 The algorithm

Webster's dictionary defines *tabu* or *taboo* as "set apart as charged with a dangerous supernatural power and forbidden to profane use or contact ..." or "banned on grounds of morality or taste or as constituting a risk ...". TS's name comes from a milder version of this definition based on the idea of imposing restrictions to prevent a stochastic search from falling into infinite loops and other undesirable behavior. TS is divided into three parts: *preliminary search*, *intensification*, and *diversification*. Preliminary search, the most important part of the algorithm, works as follows. From a specified initial configuration TS examines all neighbors and identifies the one with the highest value of the objective function. Moving to this configuration might not lead to a better solution, but TS moves there anyway; this enables the algorithm to continue the search without becoming blocked by the absence of improving moves and to climb out of local optima. If there are no improving moves (indicating a kind of local optimum), TS chooses one that least degrades the objective function. In order to avoid returning to the local optimum just visited, the reverse move now must be forbidden. This is done by storing this move, or more precisely a characterization of this move, in a data structure—the *tabu list*—often managed like a circular list, empty at the beginning and with a first-in-first-out mechanism, so that the latest forbidden move replaces the oldest one. This list contains a number s of elements defining forbidden (tabu) moves; the parameter s is called the *tabu list size*. The tabu list as described may forbid certain relevant or interesting moves, as exemplified by those that lead to a better solution than the best one found so far. In view of this, an *aspiration criterion* is introduced to allow tabu moves to be chosen anyway if they are judged to be sufficiently interesting.

Suppose for illustration that we wish to maximize a function of a binary vector of length 5, and take $(0, 1, 0, 0, 1)$ with objective function value 10 as the initial configuration. Then if we define the neighbors as those vectors obtained by one-bit flips, we obtain the following configurations $(x_1, \ldots, x_5)$, with hypothetical objective function values:

Configuration	$(1,1,0,0,1)$	$(0,0,0,0,1)$	$(0,1,1,0,1)$	$(0,1,0,1,1)$	$(0,1,0,0,0)$
Value	9	8	5.4	7.1	7.3

From the above possible moves, none leads to a better solution. But in TS we keep moving anyway, so we accept the best move among the five in the neighborhood, which is the vector $(1, 1, 0, 0, 1)$ with value 9. In order to avoid going back to the previous configuration we now mark as tabu the move that changes x_1 from 1 back to zero. So among the next five neighbors,

Configuration	$(0,1,0,0,1)$	$(1,0,0,0,1)$	$(1,1,1,0,1)$	$(1,1,0,1,1)$	$(1,1,0,0,0)$
Value	10	8.1	9.7	7.9	6.9

the first one is tabu, the rest non-tabu. The aspiration criterion is simply a comparison between the value of the tabu move and the aspiration value, which is usually the highest value found so far (in our example 10). Because the tabu move has value not larger than the aspiration value, it remains tabu; thus the available choice is among the other four. From these the one with the best solution is the third neighbor, $(1, 1, 1, 0, 1)$, with value 9.7. So now the move that changes x_3 from 1 to 0 is marked as tabu as well. Suppose that the tabu list size has been set to 4 for this example, and continue the algorithm. The next neighbors are

Configuration	$(0,1,1,0,1)$	$(1,0,1,0,1)$	$(1,1,0,0,1)$	$(1,1,1,1,1)$	$(1,1,1,0,0)$
Value	5.4	9.2	9	3	6.5

The first and the third neighbors now are tabu, with values less than the aspiration value for both, and so we have to search among the other three. Between these three moves the best one is the second neighbor, $(1, 0, 1, 0, 1)$, with value 9.2. So now the move that changes x_2 from 0 to 1 is marked as tabu as well. Going one more step produces the next neighborhood,

Vector	$(0,0,1,0,1)$	$(1,1,1,0,1)$	$(1,0,0,0,1)$	$(1,0,1,1,1)$	$(1,0,1,0,0)$
Cost	10.8	9.7	8.1	7.1	6.1

The first, second, and third moves according to the tabu list are tabu. But the first move has value larger that the aspiration value, and so its tabu status is cancelled. So the non-tabu moves now are the first, fourth, and fifth, and among these the best one is the first one, $(0, 0, 1, 0, 1)$, with value 10.8, which also replaces the best one found so far. Continuing this process for a specified number of iterations completes the preliminary search.

The next stage is intensification, which begins at the best solution found so far and clears the tabu list. The algorithm then proceeds as in the preliminary search phase. If a better solution is found, intensification is restarted. The user can specify a maximum number of restarts; after that number the algorithm goes to the next step. If the current intensification phase does not find a better solution after a specified number of iterations, the algorithm also goes to the next stage. Intensification provides a simple way to focus the search around the current best solution.

The final stage, diversification, again starts by clearing the tabu list, and sets the s most frequent moves of the run so far to be tabu, where s is the tabu list size. Then a random state is chosen and the algorithm proceeds to the preliminary search phase for a specified number of iterations. Diversification provides a simple way to explore regions that have been little visited to date. After the end of the third stage, the best solution (or k best solutions) found so far may be reported, or the entire algorithm may be repeated (always storing the k best solutions so far) either a specified number of times or until a preset CPU limit is reached.

5.2 Implementation and modifications of TS

From the above description, it is clear that there are a number of potentially crucial user-defined choices in TS, including neighborhood sizes, types of moves, tabu list structures, and aspirations conditions, and the specification of a variety of inputs (see Algorithm 5) including $maxmoves$, n_{int}, n_{impr}, n_{div} and rep. There appears to be surprisingly little advice in the literature about how to make these choices. We summarize what is known in what follows; see Fouskakis (2001) for more details.

5.2.1 Neighborhood sizes and candidate lists

When the dimension of the problem is high, a complete neighborhood examination may be computationally expensive, which suggests examining only some regions of it that contain moves with desirable features. One way of doing this is to use a *neighborhood decomposition strategy*, which breaks down the neighborhood at each iteration into coordinated subsets and uses some means (such as an aspiration threshold) of linking how the subsets are examined in order to visit less appealing regions less often. Success stories with this approach include Laguna, Barnes, et al. (1991), Semet and Taillard (1993), and Fiechter (1994). Another technique is *elite evaluation candidate lists*, in which a collection of promising moves is stored; at each iteration these elite moves are considered first, after which a subset of the usual neighborhood is examined and the worst of the elite moves are replaced by any new moves found which dominate them. From time to time, based on an iteration count or when all of the "elite" moves fail to satisfy an aspiration threshold (Glover, Glover, et al. 1986), the algorithm pauses and spends time refreshing the candidate list with good moves. A final idea that lends itself well to parallel processing (see Section 5.2.3) is a *sequential fan candidate list* (Glover et al. 1993), in which the l best alternative moves are computed at specified points in TS and a variety, or "fan," of solution trajectories is maintained independently.

5.2.2 Tabu list size

The choice of tabu list size is crucial; if its value is too small, cycling may occur in the search process, while if its value is too large, appealing moves may be forbidden, leading to the exploration of lower quality solutions and producing a larger number of iterations to find the solution desired. Empirically, tabu list sizes that provide good results often grow with the size of the problem. However, there appears to be no single rule that gives good sizes for all classes of problems.

Rules for determine s, the tabu list size, are either *static* or *dynamic*, according to whether s is fixed or permitted to vary as the TS run unfolds. Table 3 gives examples of these rules. The values of 7 and $\sqrt{p}$ (where p is the dimension of the problem) used in this table are merely for illustration, not meant as a rigid guide to implementation; but TS practitioners often use values between 7 and 20, or between $\frac{1}{2}\sqrt{p}$ and $2\sqrt{p}$, and in fact these values appear to work well for a large variety of problems (Glover 1990b).

Experience has shown that dynamic rules often provide benefits in solution quality which exceed the costs of implementing the greater flexibility, and many researchers prefer to use non-fixed values for the tabu list size. For example Taillard (1991), in order to solve the quadratic assignment problem,

Algorithm 5: Tabu Search (TS)

```
Begin;
  Randomly choose a configuration $i_{start}$, set $i := i_{start}$, and evaluate the objective function
    $f(i)$; set the aspiration value $\alpha := lo$, a small number; determine $s := Listlength$,
    the length of the tabu list; set $Move := 0$ and $i_{max} := i_{start}$;
  Repeat:
    $\boxed{\text{Preliminary Search}}$
    Add $i$ to the tabu list at position $s$; set $s := s - 1$. If $s = 0$ then set $s := Listlength$;
    set $Move := Move + 1$, $i_{nbhd} := i$, and $c_{nbhd} := low$, a small number;
    For each neighbor $j$ of $i$ do:
      If $f(j) > \alpha$ do:
        If $f(j) \geq c_{nbhd}$ then set $i_{nbhd} := j$ and $c_{nbhd} := f(j)$;
      If $f(j) \leq \alpha$ do;
        If $j$ is in the tabu list go to the next neighbor;
        Else if $j$ is non-tabu and $f(j) \geq c_{nbhd}$ then set $i_{nbhd} := j$ and $c_{nbhd} := f(j)$;
    Set $\alpha := \min(\alpha, c_{nbhd})$ and $i := i_{nbhd}$;
    If $f(i) \geq f(i_{max})$ then $i_{max} := i$;
    If $Move \neq maxmoves$ go back to Preliminary Search;
    Else go to Intensification;
    $\boxed{\text{Intensification}}$
    Repeat:
      Set $i := i_{max}$ and clear the tabu list;
      Repeat:
        Do the Preliminary Search;
        Until a better solution than $i_{max}$ is found. If no improvements after $n_{int}$ iterations
          go to the next stage;
    Until $n_{impr}$ replications;
    $\boxed{\text{Diversification}}$
    Clear the tabu list and set the $s$ most frequent moves to be tabu;
    Randomly choose a configuration $i$;
    Evaluate $f(i)$;
    Repeat:
      Do the Preliminary Search;
    Until $n_{div}$ repetitions have occurred;
  Until the whole algorithm has been repeated $rep$ times;
  $i_{max}$ is the approximation to the optimal solution;
End.
```

Table 3: *Illustrative rules for tabu list size.*

Static Rules

- Choose s to be a constant such as $s = 7$ or $s = \sqrt{p}$, where p is the problem's dimension.

Dynamic Rules

- *Simple dynamic*: Choose s to vary randomly or by systematic pattern, between bounds s_{min} and s_{max} with $s_{min} = 5$ and $s_{max} = 11$ (for example) or $s_{min} = 0.9\sqrt{p}$ and $s_{max} = 1.1\sqrt{p}$.

- *Attribute-dependent dynamic*: Choose s as in the simple dynamic rule, but determine s_{min} and s_{max} to be larger for *attributes* (move types) that are more attractive (based for example on quality or influence considerations).

selected the size randomly from the interval $[0.9\,p, 1.1\,p]$ and kept it constant for $2.2\,p$ iterations before selecting a new size by the same process. Laguna, Barnes, et al. (1991) have effectively used dynamic rules that depend on both attribute type and quality, while a class of dynamic rules based on moving gaps was also used effectively by Chakrapani and Skorin–Kapov (1993). Finally Laguna and Glover (1993) systematically varied the list size over three different ranges (small, medium and large), in order to solve telecommunications bandwidth packing problems.

5.2.3 Parallel implementations: speeding up TS

To speed up TS, again consider the case of parallel runs. According to Glover et al. (1993), concurrent examination of different moves from a neighborhood often leads to a multiplicative improvement in total runtime that is close to the *ideal speed-up*

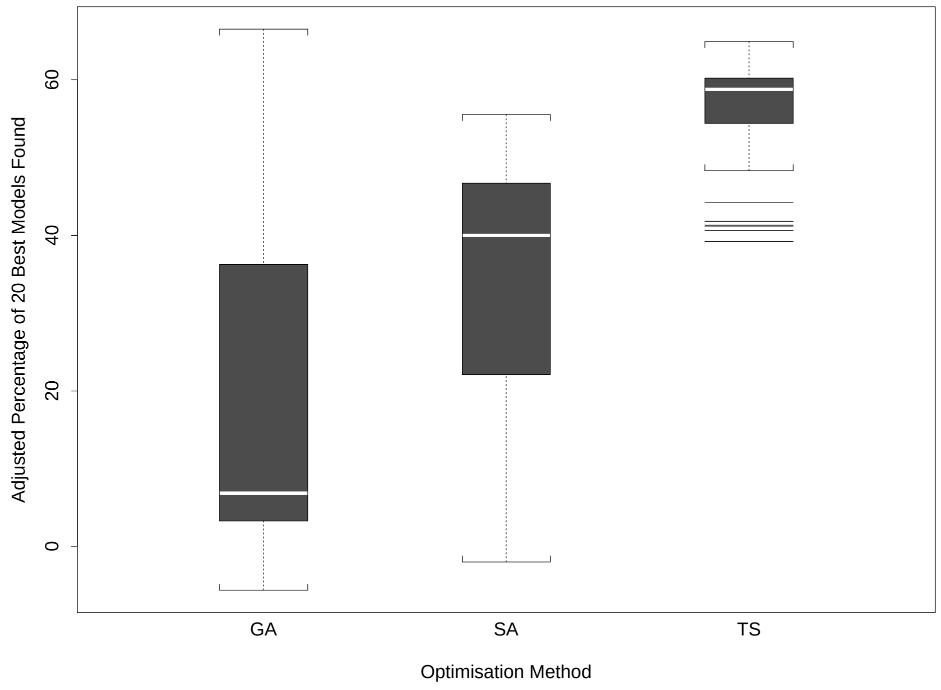
$$\frac{T_n + T_u}{\frac{T_n}{P} + T_u}, \qquad (42)$$

where P is the number of processors and T_n and T_u are the amounts of algorithmic time that can and cannot be parallelized, respectively. The basic idea is to divide the neighborhood into P parts of roughly equal size and give one part to each processor to evaluate. The processors all send messages to a master processor identifying the best moves they have found; the master node selects the overall best move and makes it, and the cycle begins again. Another natural and even simpler parallelization process is to let the processors work totally independently, each starting with its own initial configuration and perhaps also using different input settings to the tabu search. This process has no need of coordination and can be applied with any optimization problem; it has proven surprisingly effective in environments with $P \leq 20$. Taillard (1989, 1990, 1991) applied this method in three different kinds of problems with considerable success.

6 Example: Bayesian utility maximization

We have performed a large simulation experiment in the context of the Bayesian variable selection example of Section 2, to compare the performance of simulated annealing (SA), genetic algorithms (GA), and tabu search (TS) and to learn how sensitive each of these methods is to user-defined input settings in a complex optimization problem with binary configurations x. See Fouskakis (2001) for details and Draper and Fouskakis (2001) for the main conclusions; here we give a brief summary of

Figure 4: *Parallel boxplots comparing GA, SA, and TS in the 14–variable case.*



some of our findings, in the case with $p = 14$ for which Figure 1 describes the results of an exhaustive enumeration of the quality of all $2^p = 16,384$ *models* (possible subsets of predictor variables). From Figure 1 it is straightforward to pick out the k best models (with, e.g., $k = 20$) and use as a performance measure how many of these best models each optimization method can find with a limited budget of CPU time.

For each of a large variety of input settings with each of GA, SA, and TS, we restricted each method to 20 minutes of CPU time at 400 Unix MHz and recorded the mean, over 30 repetitions (using different random numbers each time), of the percentage of the actual 20 best models it was able to identify. With all three optimization methods we used an adaptive approach (details are available in Fouskakis 2001) to specifying N, the number of modeling/validation splits on which the estimated expected utility of any given model is based. With TS we varied six user inputs (the total number of repetitions of the algorithm, the maximum number N^* of random modeling/validation splits used in the adaptive method just mentioned for estimating the expected utility, the number of preliminary searches per repetition, the number of intensification searches per repetition, the maximum number of random restarts in each intensification search, and the number of diversification searches per repetition) across 49 different combinations in a partial factorial design. With SA we examined a total of 108 different choices of five user inputs (the total number of iterations, N^*, the initial and final temperatures, and the cooling schedule) in a nearly complete full-factorial design. With GA we implemented a complete full-factorial experiment over six user inputs (the total number of repetitions, N^*, the population size, the crossover strategy—simple, uniform, highly uniform—and crossover probability, elitist versus non-elitist strategies, and whether or not the population was cleared and regenerated at random at the end of each repetition) involving a total of 144 different combinations of inputs.

Figure 4 summarizes the results of this experiment by providing parallel boxplots for each optimization method across all combinations of input settings examined. Versions of GA employing elitist strategies, uniform or highly uniform crossover behavior, and smaller values of the population size were the overall winners, followed closely by versions of TS with input settings detailed in Fouskakis (2001). The performance of "vanilla" versions of SA was disappointing in this problem; Draper and Fouskakis (2001) describe an approach to making SA more "intelligent"—by accepting or rejecting a proposed move to a new model based on a simple measure of the desirability of that model in a cost-benefit sense—which exhibits greatly improved performance. While input settings can be chosen for GA with good performance in this problem, it is also true that GA performed very poorly (the worst of all three methods) with many other input settings. It is evident from Figure 4 that tabu search possesses a kind of robustness to mis-specification of input settings in this problem: the best TS performance was among the best of all optimization methods studied, and the worst was approximately as good as the median performance of "vanilla" SA.

7 Discussion

The literature comparing the performance of stochastic optimization methods such as simulated annealing (SA), genetic algorithms (GA), and tabu search (TS) has grown considerably in recent years. Three sets of authors working in different problem areas have recently made detailed comparisons of SA and GA: Franconi and Jennison (1997) in image analysis, Bergeret and Besse (1997) in problems involving the use of a sample to minimize the expectation of a random variable, and Glass and Potts (1996) in scheduling jobs in a permutation flow shop to minimize the total weighted completion time. These comparisons yield a somewhat complex picture: Franconi and Jennison find (1) that "GAs are not adept at handling problems involving a great many variables of roughly equal influence" and (2) that a GA-SA hybrid outperforms either method individually in the problem they examine; Bergeret and Besse demonstrate the superiority of SA over GA in their class of problems; and Glass and Potts find that a hybrid genetic descent algorithm performs best in their flow-shop context.

Westhead et al. (1997) compare GA and TS in problems of flexible molecular docking, concluding that GA "performs best in terms of the median energy of the solutions located" but that TS "shows a better performance in terms of locating solutions close to the crystallographic ligand conformation," suggesting that a GA-TS hybrid might outperform either method. Augugliaro et al. (1999) compare SA, GA, and TS in problems involving the reconfiguration of radial distribution electric power networks and find that TS performs best in this class of problems. Finally, Fu and Su (2000), Fudo et al. (2000), Ganley and Heath (1998), Manoharan and Shanmuganathan (1999), Sinclair (1993), and Youssef et al. (2001) compare SA, GA, and TS in problems involving the minimization of assembly time in printed wiring assembly, service restoration in electricity distribution systems, graph partitioning, structural engineering optimization, balancing hydraulic turbine runners, and the floor-planning of very large scale integrated (VLSI) circuits, respectively, in many cases without identifying any clear winners.

Two basic themes emerge from this literature: (a) the winning optimization method is highly context-specific, and (b) hybridization of competing algorithms often narrows the performance gap between methods, and sometimes yields an approach that is superior to any of the algorithms being hybridized. What has not yet emerged from this literature is an unambiguous identification of regions in the space of optimization problems in which one method clearly dominates the others, and an intuitive explanation for such dominance when it can be found.

Acknowledgements

We are grateful to Katherine Kahn for providing us with the data from the Rand health policy study, and to Merrilee Hurn, Chris Jennison, and Sylvia Richardson for comments and references that improved the paper. Membership on this list does not imply agreement with the ideas expressed here, nor are any of these people responsible for any errors that may be present.

References

Aarts EHL, Korst JHM (1989). *Simulated Annealing and Boltzmann Machines.* Chichester: Wiley.

Ackley D (1987). *A Connectionist Machine for Genetic Hill-climbing.* Dordrecht: Kluwer.

Alander JT (1992). On optimal population size of genetic algorithms. *Proceedings CompEuro 92*, 65–70. IEEE Computer Society Press.

Augugliaro A, Dusonchet L, Sanseverino ER (1999). Genetic, simulated annealing, and tabu search algorithms: three heuristic methods for optimal reconfiguration and compensation of distribution networks. *European Transactions on Electric Power*, **1**, 35–41.

Baker JE (1985). Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Davis L, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Beaty S (1991). *Instruction Scheduling using Genetic Algorithms.* Ph.D. Dissertation, Colorado State University.

Bergeret F, Besse P (1997). Simulated annealing, weighted simulated annealing and the genetic algorithm at work. *Computational Statistics*, **12**, 447–465.

Bernardo JM, Smith AFM (1994). *Bayesian Theory.* New York: Wiley.

Beyer D, Ogier R (1991). Tabu learning: a neural network search method for solving nonconvex optimization problems. *Proceedings of the International Joint Conference on Neural Networks.* Singapore: IEEE and INNS.

Bland JA, Dawson GP (1991). Tabu search and design optimization. *Computer–Aided Design*, **23**, 195–202.

Booker LB (1987). Improving search in genetic algorithms. In *Genetic Algorithms and Simulated Annealing* (Davis L, ed.). San Mateo, CA: Morgan Kaufmann.

Brandimarte P, Conterno R, Laface P (1987). FMS production scheduling by simulated annealing. *Proceedings of the 3rd International Conference on Simulation in Manufacturing*, 235–245.

Brown PJ, Fearn T, Vannucci M (1999). The choice of variables in multivariate regression: A non-conjugate Bayesian decision theory approach. *Biometrika*, **86**, 635–648.

Caruna RA, Schaffer JD (1988). Representation and hidden bias: Gray vs. binary coding for genetic algorithms. *Proceedings of the Fifth International Conference on Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Chakrapani J, Skorin–Kapov J (1993). Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, **41**, 327–341.

Cohoon JP, Hegde SU, Martin WN, Richards D (1987). Punctuated equilibria: a parallel genetic algorithm. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Committee on the Next Decade of Operations Research (1998). Operations research: the next decade. *Operations Research*, **36**, 619–637.

Connolly DT (1990). An improved annealing scheme for the QAP. *European Journal of Operational Research*, **46**, 93–100.

Cvijovic D, Klinowski J (1995). Tabu search: An approach to the multiple minima problem. *Science*, **267**, 664–666.

Davis L (1991). *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold.

Daley J, Jencks S, Draper D, Lenhart G, Thomas N, Walker J (1988). Predicting hospital-associated mortality for Medicare patients with stroke, pneumonia, acute myocardial infarction, and congestive heart failure. *Journal of the American Medical Association*, **260**, 3617–3624.

De Jong KA (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. Dissertation, University of Michigan.

de Werra D, Hertz A (1989). Tabu search techniques: a tutorial and an application to neural networks. *OR Spektrum*, **11**, 131-141.

Dowsland KA (1993). Some experiments with simulated annealing techniques for packing problems. *European Journal of Operational Research*, **68**, 389–399.

Draper D (1995). Inference and hierarchical modeling in the social sciences (with discussion). *Journal of Educational and Behavioral Statistics*, **20**, 115–147, 233–239.

Draper D (1996). Hierarchical models and variable selection. Technical Report, Department of Mathematical Sciences, University of Bath, UK.

Draper D, Fouskakis D (2000). A case study of stochastic optimization in health policy: problem formulation and preliminary results. *Journal of Global Optimization*, **18**, 399–416.

Draper D, Fouskakis D (2001). Stochastic optimization methods for cost-effective quality assessment in health. Submitted.

Draper D, Kahn K, Reinisch E, Sherwood M, Carney M, Kosecoff J, Keeler E, Rogers W, Savitt H, Allen H, Wells K, Reboussin D, Brook R (1990). Studying the effects of the DRG-based Prospective Payment System on quality of care: Design, sampling, and fieldwork. *Journal of the American Medical Association*, **264**, 1956–1961.

Eshelman L (1991). The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In *Foundations of Genetic Algorithms* (Rawlins G, ed.). San Mateo, CA: Morgan Kaufmann.

Fairley A (1991). *Comparison of methods of choosing the crossover point in the genetic crossover operation.* Department of Computer Science, University of Liverpool.

Fiechter CN (1994). A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, **51**, 243–267.

Fouskakis D (2001). *Stochastic optimisation methods for cost-effective quality assessment in health.* PhD dissertation, Department of Mathematical Sciences, University of Bath, UK (available at `www.soe.ucsc.edu/~draper`).

Fouskakis D, Draper D (1999). Review of *Tabu Search*, by F Glover and M Laguna, Amsterdam: Kluwer (1997). *The Statistician*, **48**, 616–619.

Franconi L, Jennison C (1997). Comparison of a genetic algorithm and simulated annealing in an application to statistical image reconstruction. *Statistics and Computing*, **7**, 193–207.

Fu HP, Su CT (2000). A comparison of search techniques for minimizing assembly time in printed wiring assembly. *International Journal of Production Economics*, **63**, 83–98.

Fudo H, Toune S, Genji T (2000). An application of reactive tabu search for service restoration in distribution systems and its comparison with the genetic algorithm and parallel simulated annealing. *Electrical Engineering in Japan*, **133**, 71–82.

Ganley JL, Heath LS (1998). An experimental evaluation of local search heuristics for graph partitioning. *Computing*, **60**, 121–132.

Gelfand AE, Dey DK, Chang H (1992). Model determination using predictive distributions with implementation via sampling-based methods. In *Bayesian Statistics 4*, Bernardo JM, Berger JO, Dawid AP, Smith AFM (editors), Oxford: University Press, 147–167.

Glass CA, Potts CN (1996). A comparison of local search methods for flow shop scheduling. *Annals of Operations Research*, **63**, 489–509.

Glover F (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences*, **8**, 156–166.

Glover F (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, **13**, 533–549.

Glover F (1989). Tabu search–Part I. *ORSA Journal on Computing*, **1**, 190–206.

Glover F (1990a). Tabu search–Part II. *ORSA Journal on Computing*, **2**, 4–32.

Glover F (1990b). Tabu search: A tutorial. *Interfaces*, **20**, 74–94.

Glover F, Glover R, Klingman D (1986). The threshold assignment algorithm. *Mathematical Programming Study*, **26**, 12–37.

Glover F, McMillan C (1986). The general employee scheduling problem: an integration of management science and artificial intelligence. *Computer and Operational Research*, **15**, 563–593.

Glover F, Taillard E, de Werra D (1993). A user's guide to tabu search. *Annals of Operations Research*, **41**, 3–28.

Goldberg DE (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, MA: Addison–Wesley.

Goldberg DE, Deb K (1991). A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms* (Rawlins G, ed.). San Mateo, CA: Morgan Kaufmann.

Goldberg DE, Lingle R (1985). Alleles, loci and the traveling salesman problem. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Goldberg DE, Richardson J (1987). Genetic algorithms with sharing for multimodal function optimization. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Goldstein H, Spiegelhalter DJ (1996). League tables and their limitations: Statistical issues in comparisons of institutional performance (with discussion). *Journal of the Royal Statistical Society, Series A*, **159**, 385–444.

Greene JW, Supowit KJ (1986). Simulated annealing without rejected moves. *IEEE Transactions on Computer-Aided Design*, **CAD–5**, 221–228.

Grefenstette JJ (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, **16**, 122–128.

Hadorn D, Draper D, Rogers W, Keeler E, Brook R (1992). Cross-validation performance of patient mortality prediction models. *Statistics in Medicine*, **11**, 475–489.

Hansen P (1986). The steepest ascent mildest descent heuristic for combinatorial programming. *Congress on Numerical Methods in Combinatorial Optimization*. Capri, Italy.

Hansen P, Jaumard B (1990). Algorithms for the maximum satisfiability problem. *Computing*, **44**, 279–303.

Hertz A, de Werra D (1987). Using tabu search techniques for graph coloring. *Computing*, **29**, 345–351.

Holland JH (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

Hosmer DW, Lemeshow S (1989). *Applied Logistic Regression.* New York: Wiley.

Ishibuchi H, Misaki S, Tanaka H (1995). Modified simulated annealing algorithms for the flow shop sequencing problem. *European Journal of Operational Research*, **81**, 388–398.

Jaumard B, Hansen P, Poggi di Aragao M (1991). Column generation methods for probabilistic logic. *ORSA Journal on Computing*, **3**, 135–148.

Jencks S, Daley J, Draper D, Thomas N, Lenhart G, Walker J (1988). Interpreting hospital mortality data: The role of clinical risk adjustment. *Journal of the American Medical Association*, **260**, 3611–3616.

Jog P, Suh JY, Gucht DV (1989). The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In *Proceedings of the Third International Conference on Genetic Algorithms* (Schaffer JD, ed.). San Mateo, CA: Morgan Kaufmann.

Johnson DS, Aragon CR, McGeoch LA, Schevon C (1989). Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. *Operational Research*, **37**, 865–892.

Kahn K, Brook R, Draper D, Keeler E, Rubenstein L, Rogers W, Kosecoff J (1988). Interpreting hospital mortality data: How can we proceed? *Journal of the American Medical Association*, **260**, 3625–3628.

Kahn K, Rogers W, Rubenstein L, Sherwood M, Reinisch E, Keeler E, Draper D, Kosecoff J, Brook R (1990). Measuring quality of care with explicit process criteria before and after implementation of the DRG-based Prospective Payment System. *Journal of the American Medical Association*, **264**, 1969–1973.

Kahn K, Rubenstein L, Draper D, Kosecoff J, Rogers W, Keeler E, Brook R (1990). The effects of the DRG-based Prospective Payment System on quality of care for hospitalized Medicare patients: An introduction to the series. *Journal of the American Medical Association*, **264**, 1953–1955.

Kapsalis A, Smith GD, Rayward–Smith VJ (1993). Solving the graphical Steiner tree problem using genetic algorithms. *Journal of the Operational Research Society*, **44**, 44.

Keeler E, Kahn K, Draper D, Sherwood M, Rubenstein L, Reinisch E, Kosecoff J, Brook R (1990). Changes in sickness at admission following the introduction of the Prospective Payment System. *Journal of the American Medical Association*, **264**, 1962–1968.

Kirkpatrick S, Gelatt CD, Vecchi MP (1983). Optimization by simulated annealing. *Science*, **220**, 671–680.

Kitano H (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, **4**, 461–476.

Laguna M, Barnes JW, Glover F (1991). Tabu search methods for a single machine scheduling problem. *Journal of Intelligent Manufacturing*, **2**, 63–74.

Laguna M, Glover F (1993). Bandwidth packing: A tabu search approach. *Management Science*, **39**, 492–500.

Laguna M, Gonzalez-Velarde JL (1991). A search heuristic for just–in–time scheduling in parallel machines. *Journal of Intelligent Manufacturing*, **2**, 253–260.

Lindley DV (1968). The choice of variables in multiple regression (with discussion). *Journal of the Royal Statistical Society, Series B*, **30**, 31–66.

Lundy M, Mees A (1986). Convergence of an annealing algorithm. *Mathematical Programming*, **34**, 111–124.

Malek M, Guruswamy M, Pandya M, Owens H (1989). Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, **21**, 59–84.

Manoharan S, Shanmuganathan S (1999). A comparison of search mechanisms for structural optimization. *Computing Structures*, **73**, 363–372.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, **21**, 1087–1091.

Meyer TP, Packard NH (1992). Local forecasting of high-dimensional chaotic dynamics. In *Nonlinear Modeling and Forecasting* (Casdagli M, Eubank S, ed.). Reading, MA: Addison-Wesley.

Michalewicz Z, Janikow CZ (1991). Genetic Algorithms for numerical optimization. *Statistics and Computing*, **1**, 75–91.

Montana DJ, Davies LD (1989). Training feedforward networks using genetic algorithms. *Proceedings of the International Joint Conference on Artificial Intelligence.* San Mateo, CA: Morgan Kaufmann.

Mühlenbein H, Gorges-Schleuter M, Krämer O (1988). Evolution algorithms in combinatorial optimization. *Parallel Computing*, **7**, 65–85.

Murata T, Ishibuchi H, Tanaka H (1996). Genetic algorithms for flow-shop scheduling problems. *Computers and Industrial Engineering*, **40**, 1061–1071.

Ogbu FA, Smith DK (1990). The application of the simulated annealing algorithm to the solution of the n/m/Cmax flowshop problem. *Computers and Operational Research*, **17**, 243–253.

Oliveira S, Stroud G (1989). A parallel version of tabu search and the path assignment problem. *Heuristics for Combinatorial Optimization*, **4**, 1–24.

Pettey CB, Leuze MR, Grefenstette JJ (1987). A parallel genetic algorithm. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Rawlins GJE (1991). *Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.

Reeves CR (1992). A genetic algorithm approach to stochastic flowshop sequencing. *Proceedings of the IEEE Colloquium on Genetic Algorithms for Control and Systems Engineering.* Digest No. 1992/106, London: IEEE.

Reeves CR (1995). *Modern Heuristic Techniques for Combinatorial Problems.* London: McGraw-Hill.

Schaffer JD, Caruana RA, Eshelman LJ, Das R (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms* (Schaffer JD, ed.). San Mateo, CA: Morgan Kaufmann.

Schultz–Kremer S (1992). Genetic algorithms for protein tertiary structure prediction. In *Parallel Problem Solving from Nature 2* (Männer R, Manderick B, ed.). North-Holland.

Sechen C, Braun D, Sangiovanni-Vincetelli A (1988). Thunderbird: A complete standard cell layout package. *IEEE Journal of Solid-State Circuits*, **SC-23**, 410–420.

Semet F, Taillard E (1993). Solving real–life vehicle routing problems efficiently using tabu search. *Annals of Operations Research*, **41**, 469–488.

Sinclair M (1993). A comparison of the performance of modern heuristics for combinatorial optimization on real data. *Computing and Operations Research*, **20**, 687–695.

Sirag DJ, Weisser PT (1987). Towards a unified thermodynamic genetic operator. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Skorin–Kapov J (1990). Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing*, **2**, 33–45.

South MC, Wetherill GB, Tham MT (1993). Hitchhiker's guide to genetic algorithm. *Journal of Applied Statistics*, **20**, 153–175.

Stander J, Silverman BW (1994). Temperature schedules for simulated annealing. *Statistics and Computing*, **4**, 21–32.

Suh JY, Van Gucht D (1987). Incorporating heuristic information into genetic search. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (Grefenstette JJ, ed.). Hillsdale, NJ: Lawrence Erlbaum Associates.

Syswerda G (1989). Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms* (Schaffer JD, ed.). San Mateo, CA: Morgan Kaufmann.

Taillard E (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, **47**, 65–74.

Taillard E (1991). Robust tabu search for the quadratic assignment problem. *Parallel Computing*, **17**, 443–455.

Talbi EG, Bessière P (1991). A parallel genetic algorithm applied to the mapping problem. *SIAM News*, July 1991, 12–27.

Tovey CA (1988). Simulated simulated annealing. *AJMMS*, **8**, 389–407.

Ulder NLJ, Aarts EHL, Bandelt HJ, Van Laarhoven PJM, Pesch E (1991). Genetic local search algorithms for the traveling salesman problem. In *Parallel Problem Solving from Nature* (Schwefel HP, Manner R, ed.). Berlin: Springer.

Vakharia AJ, Chang YL (1990). A simulated annealing approach to scheduling a manufacturing cell. *Naval Research Logistics*, **37**, 559–577.

Van Laarhoven PJM, Aarts EHL (1988). *Simulated Annealing and Boltzmann machines.* Chichester: Wiley.

Weisberg S (1985). *Applied Linear Regression*, second edition. New York: Wiley.

Westhead DR, Clark DE, Murray CW (1997). A comparison of heuristic search algorithms for molecular docking. *Journal of Computer-Aided Molecular Description*, **11**, 209–228.

Whitley D (1989). The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms* (Schaffer JD, ed.). San Mateo, CA: Morgan Kaufmann.

Whitley D (1992). *Foundations of Genetic Algorithms 2.* San Mateo, CA: Morgan Kaufmann.

Widmer M (1991). Job shop scheduling with tooling constraints: a tabu search approach. *Journal of Operation Research Society*, **42**, 75–82.

Youssef H, Sait SM, Adiche H (2001). Evolutionary algorithms, simulated annealing and tabu search: a comparative study. *Engineering Applications of Artificial Intelligence*, **14**, 167–181.