Stochastic Optimisation Methods

Aim

We are seeking the maximum (or minimum) of $f(x)$, $x \in \mathcal{X}$, with respect to x, where x is either a number or a vector. The set $\mathcal{X}$ consist of N elements, for a finite positive integer N.

E.g. Consider a multiple linear regression problem with p potential predictor variables. A fundamental step in regression is selection of a suitable model. Given a dependent variable Y and a set of candidate predictors $x_1, \ldots, x_p$, we must find the best model of the form $Y = b_0 + \sum_{j=1}^{s} b_{i_j} x_{i_j} + \epsilon$, where $\{i_1, \ldots, i_s\}$ is a subset of $\{1, \ldots, p\}$. The goal could be the minimisation of the Akaike information criterion (AIC), where $AIC = Nlog\{RSS/N\} + 2\{s+2\}$, where N is the sample size, s the number of predictors in the model and RSS the residual sum of squares.

We can also use an indicator γ_j, taking the value 1 if variable j is included in the model and 0 otherwise, and thus in this case we would like to minimise AIC over $\mathcal{X} = \{0, 1\}^p$.

In order to solve problem like the one described above we use **Stochastic** (or Combinatorial) Optimisation Techniques, which also sometimes are called **heuristics**. The two primary features of such techniques are (a) iterative improvement of a current candidate solution and (b) limitation of the search to a local neighborhood at any particular iteration.

Local Search

- Begin;
- Choose a random configuration x_{start};
- Set $x := x_{start}$;
- Repeat:
- Generate a new configuration x' from the neighbourhood of x;
- If $f(x') \geq f(x)$ then $x := x'$;
- Until $f(x') \leq f(x)$ for all x' from the neighbourhood of x;
- End.

The disadvantages of local search algorithms can be formulated as follows:

- By definition, local search algorithms terminate in a local maximum and there is generally no information as to the amount by which this local maximum deviates from a global maximum;

- The obtained local maximum depends on the initial configuration, for the choice of which generally no guidelines are available; and

- In general, it is not possible to give an upper bound for the computation time.

To avoid some of the above mentioned disadvantages, one might think of a number of alternative approaches:

- Execution of the algorithm for a large number of initial configurations, say M, at the cost of an increase in computation time; for $M \to \infty$, such an algorithm finds a global maximum with

probability 1, if only for the fact that 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 generation mechanism, in order to be able to "jump out" of the local maxima corresponding to the simple generation mechanism. To choose the more complex 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.

Tabu Search

Tabu search (TS) is a "higher level" heuristic procedure for solving large optimisation problems, proposed by Glover (1989).

TS has three phases: **preliminary search**, **intensification**, and **diversification**.

During **preliminary search** TS is similar to some other optimisation methods in that, whatever point x in the input space you are currently at, you evaluate the criterion function at all the **neighbours** $\mathcal{N}$ of x and find the new point x' that is best in $\mathcal{N}$, but TS differs from many other methods in that you move to x' **even if it is worse than** x.

Repeating this idea creates the possibility of endlessly cycling back and forth between x and x'; to avoid this TS uses the idea of a **tabu list** of forbidden moves, so that (e.g.) once the move $x \to x'$ has been made the reverse move $x' \to x$ is forbidden for at least the next s moves.

One potential problem with the tabu list is that **it may forbid certain relevant or interesting moves**, for example those that lead to a better x than the best one found so far.

Consequently, an **aspiration criterion** is introduced to allow moves that would otherwise be tabu to be chosen anyway, if they are judged to be worthwhile.

In the second (**intensification**) part of the search, you (a) start with the best solution found so far (which is always stored throughout the entire algorithm), (b) clear the tabu list, and (c) proceed as in the preliminary search for a specified number of moves. (This step can be **restarted randomly** a given number of times.)

Finally, in the **diversification** phase, you again clear the tabu list, and set the s most frequent moves of the run so far to be tabu.

Then you choose a random x to move to and **proceed as in the preliminary search phase** for a specified number of iterations.

Some implementation issues of TS

- **Neighbourhood sizes and candidate lists**

1) **Moves**: E.g in regression (1 bit flip vs. 2 bit swap) 2 **Neighbourhood sizes**: dynamic vs. static

- **Tabu list size**

if its value is too small, cycling may occur, while if its value 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.

Rules for determine s, the tabu list size, are classified as *static* or *dynamic*. Static rules choose a value for s that remains fixed throughout the run; dynamic rules allow the value of s to vary. The values of 7 and $\sqrt{p}$ (where p is the dimension of the problem) often used.

Simulated Annealing

Simulated annealing (SA) was proposed by Kirkpatrick et al (1983) as a technique for discrete optimisation dates back to the early 1980s. It was heralded with much enthusiasm as it appeared to be both simple to implement and widely applicable, and as a result of articles in popular scientific journals researchers from a wide variety of disciplines experimented with it in the solution of their own problems.

The ideas that form the basis of SA were first published by Metropolis et al (1953) in an algorithm to simulate the cooling of material in a heat bath—a process known as annealing. If solid material is heated past its melting point and then cooled back into a solid state, the structural properties of the cooled solid depend on the rate of cooling. The annealing process can be simulated by regarding the material as a system of particles. Essentially, the Metropolis algorithm simulates the change in energy of the system when subjected to a cooling process, until it converges to a steady "frozen" state. Thirty years later Kirkpatrick et al (1983) suggested that this type of simulation could be used to solve optimisation problems.

The algorithm is as follows:

- Choose a **random initial configuration** x_{start}. Set $x = x_{start}$;

- Set the initial **temperature** T_0;

- **Repeat:**

  - Choose at random a **neighboring** configuration x';

    - If $f(x') > f(x)$ then set $x = x'$;

    - else set $x = x'$ with probability

$$\exp\left\{\frac{f(x') - f(x)}{T}\right\}.$$

    - let T decrease according to some **schedule**;

- Until the **final temperature** T_f is reached.

Some Implementation Issues of SA

- **Neighbourhood sizes and candidate lists**

1) **Moves**: E.g in regression (1 bit flip vs. 2 bit swap)

2 **Neighbourhood sizes**: dynamic vs. static

- **Upper and lower limits for T**

Usually $T_0 = 1$ and $T_f = 0.1$

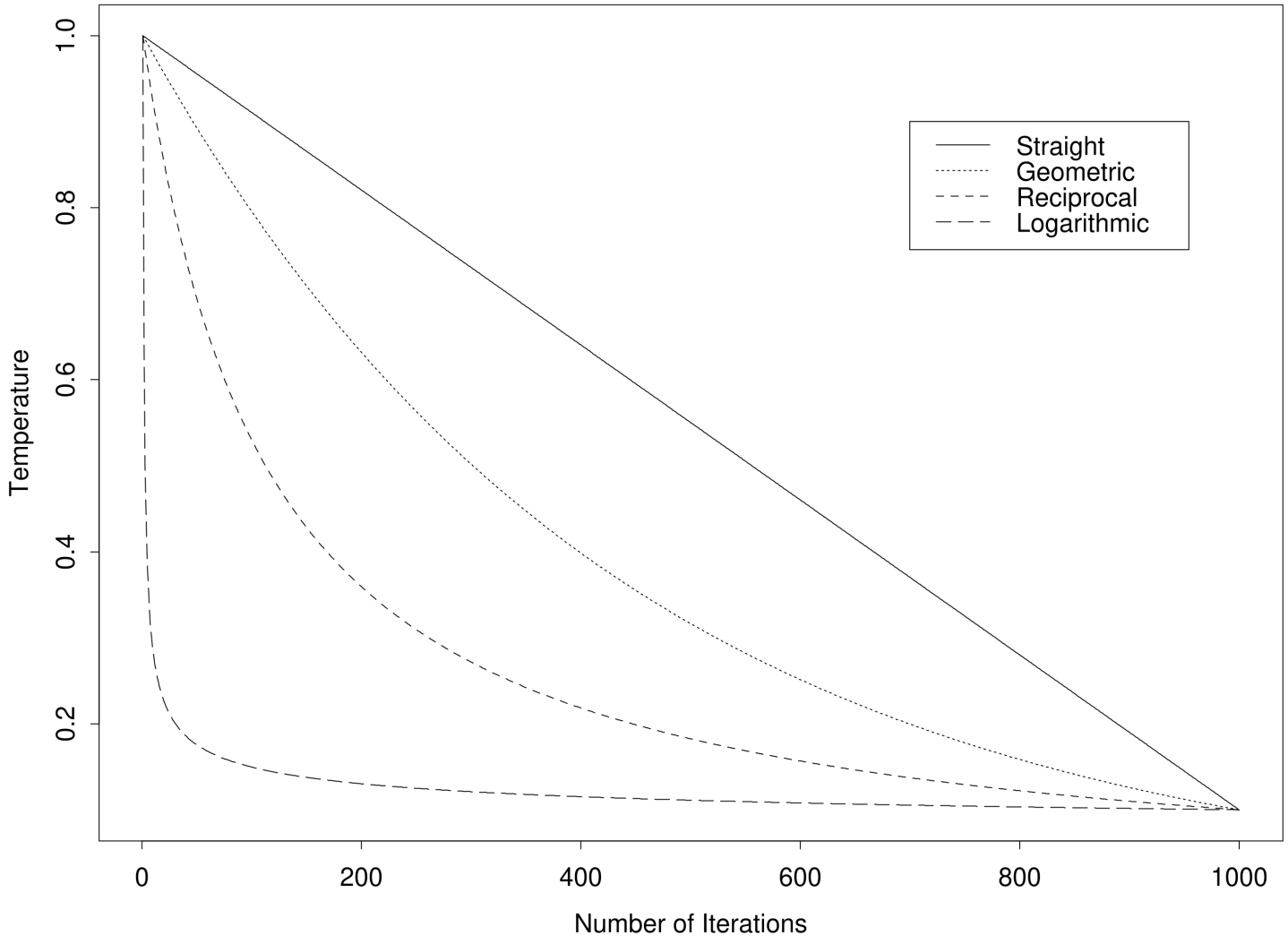
- **Temperature Schedule**

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

where M is the total number of iterations.

Some Implementation Issues of SA

The four temperature schedules in the previous Table, with $T_0 = 1.0, T_f = 0.1$, and with a total number of iterations $M = 1,000$.



Genetic Algorithms

A method first proposed by Holland (1975), based on ideas from Biology.

All living organisms consist of cells, and each cell contains the same set of one or more strings of DNA called **chromosomes**. The chromosomes are also be divided into **genes**, functional blocks of DNA, each of which encodes a particular protein. For instance, you can think that a gene can represent the eye colour of the organism. Then the different settings that this eye colour can take (e.g., brown, blue, etc.), are called **alleles**. The position of each gene on the chromosome is called the **locus**. The organisms may have multiple chromosomes in each cell. The complete collection of genetic material is called the organism's **genome**. Two individuals that have identical genomes are said to have the same **genotype**. The genotype gives rise, under fetal and later development, to the organism's **phenotype**–its physical and mental characteristics. In order now to produce a new off-spring, a **crossover** operation occurs. In each parent, genes are exchanged between each pair of

chromosomes to form a **gamete** (a single chromosome), and then gametes from the two parents pair up to create a full set of chromosomes. Offspring also are subject to the **mutation** operator, in which single **nucleotides** (elementary bits of DNA) are changed from parent to off-spring. The **fitness** of the organism, finally, is defined as the probability that the organism will live to reproduce, or as a function of the number of off-spring the organism has.

In GA the term chromosome typically refers to a candidate solution to a problem, and is most often is simply a binary 0–1 string. The genes are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution. The allele is either 0 or 1. The crossover operation is simply an exchange of sections of the two parents' chromosomes, while mutation is a random modification of the chromosome which can be done by flipping the bit at a randomly chosen locus.

The algorithm is as follows:

• Randomly generate an even number n of models for the initial population (using fair coin–tossing) and compute their **fitness**

• Select n models for the next population with replacement from the initial population, with probability proportional to the fitness of every model.

• Consider the models of the previous population in pairs and perform the **crossover** operation with probability p_c: Take every pair and if crossover occurs, generate an integer k from $U(1, l-1)$, where l is the number of variables, and the last $(l-k)$ elements of each model are exchanged to create 2 new models.

• Consider the previous population and perform the **mutation** operation for each variable of each model with probability p_m: If mutation occurs then flip the value of the variable from 0 to 1 or vice versa.

• Store best result and continue the algorithm for a specified number of iterations.

Some Implementation Issues of SA

- **Population size**

If too small we are not exploring the space enough, if too big there is a big computational cost. If our problem is binary usually $p \leq n \leq 2p$, otherwise usually $2p \leq n \leq 20p$, where p is the dimension of our problem.

- **Crossover probability**

Usually depends on n (population size); if n small (around 30) we use a large value of around 0.9. For populations of size around 50 we use a value around 0.5 and finally for large populations of size around 80 we use a value of 0.3.

- **Mutation probability**

Usually a small value, like 0.1 is used.

- **Fitness function**

Usually the fitness is the function g that we want to optimise or a monotone increasing function of

your objective function g. Also some researchers prefer their fitness function to take values in the interval $[0, 1]$, and therefore one solution would be to use

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

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

- **Crossover Operator**

So far we have seen the simple crossover operator. where we randomly choose a single position and we exchange the elements of the two parents. Many researchers have claimed big 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 following offspring:

$$0\ 1\ 0\ 0\ 0\ 1\ 0\ 0$$
$$1\ 0\ 1\ 0\ 1\ 1\ 1\ 0$$

by taking the first two and the last two elements from the one parent and the rest from the other each time.

The operator that has received the most attention in recent years is the *uniform crossover*. Suppose we have the following strings:

1 0 0 0 1 0 1 0

0 1 0 1 0 0 1 1

Then for each position randomly (with probability usually 0.5) pick each bit from either of the two parent strings. If you want to produce two offspring you can do the above twice. So for example we can produce the following offspring:

0 0 0 1 0 0 1 0

1 1 0 0 1 0 1 1

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 version that the *CHC Adaptive Search Algorithm*

(Eshelman, 1991) uses, which I will call **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. So for example if we have again the following parents,

1 0 0 0 1 0 1 0

0 1 0 1 0 0 1 1

we can see that the non-matching alleles are the first, second, fourth, fifth, and eighth, which are five in total. So we are going to cross three of them randomly, (say) the first, second and fourth, and so the producing children are going to be

0 1 0 1 1 0 1 0

1 0 0 0 0 0 1 1

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

- **Elitist Strategy**

Compare the parents with the offspring, and instead of copying the offspring directly to the new population, copy the two best among the four to the new population.

No binary Problems

E.g. traveling salesman problem with p=9.

A) Mutation

Randomly exchange two alleles in the chromosome. From a parent chromosome '752631948' create offspring '572631948' by exchanging the first two alleles.

B) Crossover

Problem:

From two parents chromosomes '752631948' and '912386754' and a crossover point between the second and the third loci, standard crossover would produce offspring '752386754' and '912631948'; that are both invalid.

- **Order Crossover**

A random collection of loci is chosen, e.g. 4, 6, 7. In the first parent these loci have alleles 6, 1

and 9 respectively. We find these alleles in the second parent ('**238*754') and we rearrange them in the same order that they are appearing in the first parent, creating like that the first offscpring ('612389754'). We reverse the roles of the two parents and now the alleles of the 4th, 6th and 7th position are 3, 6 and 7 respectively. Again we find these alleles in the first parent ('*52**1948') and we rearrange them in the same order that they are appearing in the second parent, creating like that the second offscpring ('352671948').

- **Edge - Recombination Crossover**

The previous operation has the undesirable tendency to destroy links between loci (cities in the traveling salesman problem) in the parents tours, and therefore they are behaving like mutation operations. The Edge - Recombination Crossover has been proposed to produce offspring (one only) that contain only links presents in at least one parent.

1) We first construct an edge table that stores all the links that lead into and out of each city in either parent (parent1 = '752631948' and parent2='912386754'. Note that the number of links

into and out of each city in either parent will be at least two and no more than four. Also recall that a tour returns to its starting city, so for example the first parent justifies listing 7 as link from 8.

2) To begin creating an offspring, we choose between the initial cities of the 2 parents the one with the fewer links. If they have the same number of links then choose randomly. In our example the choice is 9 and the offspring is '9********'

3) From the links of 9 we choose the one with the fewer links. In our case 1 and 4 have exactly 3 links so randomly we choose 4. The offspring is now '94*******'.

4) Continue like above.

The final offspring is '945786312'.

Step1		Step2		Step3	
City	Links	City	Links	City	Links
1	3, 9, 2	1	3, 2	1	3, 2
2	5, 6, 1, 3	2	5, 6, 1, 3	2	5, 6, 1, 3
3	6, 1, 2, 8	3	6, 1, 2, 8	3	6, 1, 2, 8
4	9, 8, 5	4	8, 5	4	Used
5	7, 2, 4	5	7, 2, 4,	5	7, 2
6	2, 3, 8, 7	6	2, 3, 8, 7	6	2, 3, 8, 7
7	8, 5, 6	7	8, 5, 6	7	8, 5, 6
8	4, 7, 3, 6	8	4, 7, 3, 6	8	7, 3, 6
9	1, 4	9	Used	9	Used
'9********		'94*******		'945******	

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 if our population size is n, we draw the best n unique individuals from the parent and offspring populations 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 certain Hamming distance (say d) away from each other. This form of "incest prevention" is designed to promote diversity. Usually we start with $d = \frac{p}{4}$, where p is the length of the string. If the new population is exactly the same as the previous one, we decrease d and we rerun the algorithm. When d becomes negative the result is the *divergence procedure* in which we replace the current population with n copies of the best member of the previous population, and for all but one member of the current population we flip $r \times p$ bits at random where r is the divergence rate (for instance the compromise value 0.5). We replace d by $d = r(1-r)p$ and restart the algorithm.