Biologically Inspired Methods: Cheatsheet

Vishnu

Saturday 20th April, 2019

1 Exploitation vs. Exploration

All guided searching algorithms try to strike a balance between these. *Exploitation* is taking known factors and using them to intensify them search: for example, on finding a good potential solution, search the area around it for better solutions. This allows us to follow trends in the search space, but often leads to being stuck in local minima.

Exploration involves diversifying the search by visiting previously unknown areas and trying different values to discover potentially better solutions. Doing this too much leads to an effectively random search, but in moderation this allows us to identify and break out of local minima.

2 Genetic Algorithms

Genetic Algorithms work by evolving a pool of candidate solutions, optimising them in the process till it eventually converges to an optimal enough solution. Optimising *usually* means minimising, unless otherwise specified, and this is the assumption throughout these notes. To convert from a maximisation to a minimisation problem, just multiply the cost function by -1 Each solution is represented as a **chromosome**, which contains values for the variables of the given problem (each variable is known as a **gene**). For example, if we're optimising a function of the form $f(x, y)$, then each chromosome contains two genes. **Binary Genetic Algorithms** represent chromosomes as a binary string, **Continuous Genetic Algorithms** represent them as a tuple (or list or array) of genes.

The general form of a genetic algorithm has 4 main parts:

1. Natural Selection
2. Pair Selection
3. Crossover
4. Mutation

2.1 Binary Strings

For the **Binary Genetic Algorithm**, we also have to determine how to convert to and from binary strings. Using binary strings is mostly historical, back when the memory used by each chromosome was important. It's also fairly simple to implement.

2.1.1 Binary String Length

The overall length of the binary string is dependent on the number of possible values our gene can take. For example: for an equation $f(x, y)$, if x has 100 possible values and y has 30, the overall length of the binary string representation will be $7 + 5 = 12$.
Normally though, we're given a range and precision that our genes must adhere to. Rephrasing our earlier example: $0 \leq x \leq 1$ with a precision of 2 decimal places, $5 \leq y \leq 8$ with a precision of 1 decimal place. To convert from these to the above, we use the following equation:

$$Number\_of\_Values = \frac{x_{hi} - x_{lo}}{10^{-d}} \tag{1}$$

This equation is the same as saying $integer\_range * 10^{decimal\_places}$. Now that we have the number of possible values a gene can take, we need to find the minimum length of a binary string that can represent that many distinct values. A binary string of length **n** can represent $2^n - 1$ distinct values.

For example: using the above equation, we see there are 30 distinct values for y. $2^4 - 1$ is 15, $2^5 - 1$ is 31. Therefore we choose a binary string of length 5 to represent y. We could have also calculated this as $ceil(log_2(30))$.

The above can be also be summarised as:

$$\frac{x_{hi} - x_{lo}}{10^{-d}} \leq 2^m - 1 \tag{2}$$

Where m is the required length. To get the full string, do this for each variable and add them together.

2.1.2 Binary Encoding and Decoding

Our binary representation roughly contains one unique string for each possible value. It's not going to be a perfect match though, as binary can't perfectly split decimal. To ease in the conversion, we calculate the **Binary Value**:

$$\frac{x_{hi} - x_{lo}}{2^m - 1}$$

This gives us the decimal value of a 1 in the binary string, and we'll represent it as b. Representing the binary string as BS and the decimal value as D, the conversions are:

$$Encoding : BS = binary(round(\frac{D - x_{lo}}{b})) \tag{3}$$

$$Decoding : D = x_{lo} + decimal(BS) * b \tag{4}$$

With these, x_{lo} is represented as $0 \ldots 0$ and x_{hi} as $1 \ldots 1$ for every gene. The complete binary string for a chromosome is just the concatenation of the strings for each gene.

2.1.3 Grey Codes

One problem with binary strings is adjacent numbers can have very different representations. For example: 3 and 4 have a difference of 1 in decimal, but their binary representations have every bit different (011 and 100). We can quantify this as the **Hamming Distance**, which is the number of bits that are different between two binary numbers (e.g 011 and 100 have a h.d of 3, 001 and 101 have a h.d of 1).

Grey codes are an alternate representation that guarantee consecutive numbers to have a Hamming Distance of 1. Representing the binary string as b and the grey code as g, both with length n, the conversions are as follows:

Binary to Grey:

$$g[i] = \begin{cases} b[i], & \text{if } i = n \\ b[i]\, XOR\, g[i+1], & \text{otherwise} \end{cases} \tag{5}$$

Grey to Binary:

$$b[i] = \begin{cases} g[i], & \text{if } i = n \\ b[i+1]\, XOR\, g[i], & \text{otherwise} \end{cases} \tag{6}$$

2.2 Natural Selection

This part of the algorithm determines which chromosomes are carried forward to the next generation.

2.2.1 X_{rate}

Only the best survive. The top (lowest cost) N_{keep} chromosomes are chosen, and the rest discarded. This method guarantees a fixed amount from each generation are kept, though this can be varied by changing X_{rate} (a user-defined percentage) at each iteration. With N_{pop} as the size of the population, N_{keep} can be calculated as:

$$N_{keep} = N_{pop} * X_{rate}$$

2.3 Thresholding

Only chromosomes with a score below a given threshold survive. This has the advantage of not having to sort and rank the chromosomes, but there is no guarantee on how many chromosomes survive. This can be fixed by changing the threshold at each iteration, either to an average or to a reasonable value.

2.4 Pair Selection

This part of the algorithm determines how chromosomes are paired up for crossovers (creating new chromosomes). This pairing up is important as it can influence how genes spread throughout the population, and can prevent good genes being overshadowed by poor ones. All of the below are independent of the cost function.i.e they don't require it to be of a certain form, such as differentiable.

2.4.1 Adjacency

Just pair adjacent chromosomes from top to bottom. If the population is sorted, this pairs chromosomes according to their ranks, which produces chromosomes without introducing much diversity (exploitation as opposed to exploration). It's very simple to implement, and every chromosome is used in crossovers.

2.4.2 Random

Randomly choose pairs. To generate the pair, a chromosome is chosen randomly, twice. This gives us unmatched diversity (exploration over exploitation), which gives a higher chance to find better quality offspring. Again very simple to implement, but for a completely random selection there's a chance we choose the same chromosome twice in a single pair.

2.4.3 Rank-Weighted Roulette

Choose pairs with probabilities according to their ranks. To generate the pair, two chromosomes are chosen according to the probabilities and paired together. If random can be taken as given each chromosome an equal chance to be chosen, Rank Weighted Pairing gives each chromosome a chance proportional to their rank in the population, which requires us to sort the population. This favours chromosomes with better costs. With a population of size N_{keep}, this probability for chromosome C_n of rank n is calculated as:

$$P(C_n) = \frac{N_{keep} + 1 - n}{\sum_{i=1}^{N_{keep}} i} \tag{7}$$

This gives probability $1/sum(N_{keep})$ to the last chromosome and $N_{keep}/sum(N_{keep})$ to the first. This gives a large difference in probability between adjacent chromosomes, regardless of their actual weights. For small populations, this gives a very skewed probability.

2.4.4 Cost-Weighted Roulette

Choose pairs with probabilities according to their weights. This doesn't need us to sort the population, but we do need to account that some costs may be negative. This is fixed by normalising the costs:

$$NCost(C_n) = Cost(C_n) - C_{N_{keep}+1} + 1 \tag{8}$$

$C_{N_{keep}+1}$ is the cost of the best chromosome not chosen. For X_{rate} selection this is obvious, for threshold selection or where $N_{keep} = 0$, instead use 2 * the cost of the lowest chromosome. As this value is guaranteed to be $\geq$ the lowest chromosome, subtracting it guarantees that all Ncosts are < 0. With the normalised costs, probability for each chromosome is calculated as:

$$P(C_n) = \frac{NCost(C_n)}{\sum_{i=1}^{N_{keep}} NCost(C_n)} \tag{9}$$

This gives a more accurate reflection of the relationships between the weights of the chromosomes, but is more computationally expensive.

2.5 Tournament Selection

A small subset of chromosomes is randomly chosen, and the best performing chromosome from this subset. This is done twice to get a pair, and the size of each subset is normally 2-3. A variation involves taking a larger subset, ranking them and then performing *rank-weighted roulette* to choose two chromosomes.

Both variations favour chromosomes with better costs. Also, since in each instance only a trivial subset has to be sorted, it reduces computation on sorting and thus works well for problems with large populations.

2.6 Crossover

Once the parents are selected, to generate offspring we take parts from each and combine them to make new chromosomes. Other than a few exceptions these don't generate new values: just re-arrange the existing ones. Repeatedly doing just crossovers only covers a limited section of the search space, so isn't sufficient to find an optimal solution. Crossover methods can be combined and altered: for example, a Single Point crossover followed by Blending, or only considering a subset of indices for Extrapolation.

Only the first three crossovers are applicable to the Binary Genetic
Algorithm. Crossovers for ordering problems are detailed in a separate section.

2.6.1 Single Point

This breaks each parent into two parts, and creates offspring by swapping the second part for each. A random number $0 \leq n < L$ is generated, and used to split the chromosomes. L is the length of the binary string / number of genes for binary and continuous genetic algorithms respectively. Using slice notation, this can be shown as:

$$Child_1 = Parent_1[0 : n] + Parent_2[n : end]$$
$$Child_2 = Parent_2[0 : n] + Parent_1[n : end]$$

2.6.2 Double Point

This breaks each parent into 3 parts, and swaps the middle parts for each. This middle section is denoted by two crossover points, which are the start and end of the section respectively. These follow $0 \leq n_1 < n_2 \leq L$. Using slice notation, this can be shown as:

$$Child_1 = Parent_1[0 : n_1] + Parent_2[n_1 : n_2] + Parent_1[n_2 : end]$$
$$Child_2 = Parent_2[0 : n_1] + Parent_1[n_1 : n_2] + Parent_2[n_2 : end]$$

2.6.3 Uniform Crossover

This uses a variable $\mu \in (0, 1)$ to determine how much crossover occurs. For each index of the binary string/gene tuple, generate a random number n $\in (0, 1)$. If $n < \mu$, the bit/gene in that index is swapped in the children, otherwise are directly copied from the parent. The addition of μ gives us a tuning parameter, but if it's too high or low effectively no crossover will happen (no swaps or all swaps).

2.6.4 Blending

This uses a variable $\beta \in (0, 1)$ to combine the genes from the parents (P) into new values as below:

$$Child_1[i] = \beta * P_1[i] + (1 - \beta) * P_2[i]$$
$$Child_2[i] = \beta * P_2[i] + (1 - \beta) * P_1[i]$$

These values are in the range (P_1[i], P_2[i]), so while these generate new values they are still limited to a subset of the search space. β can change from generation to generation, etc.

2.6.5 Extrapolation

This also uses a variable β, but there's no limit on its value. Children are generated using the below:

$$\begin{aligned} Diff_i &= abs(P_1[i] - P_2[i]) \\ Child_1[i] &= P_1[i] - \beta * Diff_i \\ Child_2[i] &= P_2[i] + \beta * Diff_i \end{aligned}$$

With some re-arrangement these are the same equations as Blending, but as there's no limit on β this can explore the entire search space without mutation. β can change from generation to generation, and if needed we can artificially limit the range of values generated.

2.7 Mutation

Mutation is the process that actually drives evolution: by changing the chromosomes into new values, new solutions can be evaluated and used. It is possible to have an algorithm that consists entirely of mutations: this would perform very similar to random walk or (1+1)ES. Crossovers move the genes into potentially better solutions, mutations discover new solutions.

2.7.1 How many Mutations?

Mutation is determined using a mutation rate, μ. To prevent good solutions from being mutated, we can designate the top n chromosomes as *elite*, and not consider them for mutation. n can be left as 0, in which case any chromosome can be mutated.

With N_{pop} as the size of the population, the standard formula is:

$$M = (N_{pop} - n) * \mu * N_{bits} \tag{10}$$

N_{bits} is the length of the binary string for a chromosome for BGA, and the number of genes in a chromosome for CGA.

2.7.2 Binary Mutation

Disregarding the top n chromosomes, choose M bits randomly across the population and flip them (0→1, 1→0).

2.7.3 Numeric Mutation

This uses a variable σ to control how much a gene mutates. Disregarding the top n chromosomes, choose M genes randomly and apply:

$$Gene = Gene + \sigma * N_n(0, 1)$$

N_n is a random number generator, but generates numbers according to a standard distribution.