Genetic Algorithms

A few more words

We have seen how the Hamming Cliff problem can disrupt the convergence of a genetic algorithm when we use a bit-vector to represent one or more integers or reals from a range.

The obvious (and successful) alternative is to represent each solution as a set of real numbers, and forget about the bit-vector representation.

Consider the problem of minimizing the surface area of a cylinder, subject to a requirement that the volume be at least some given value \( k \) (this standard problem is taken from the literature but it will become relevant if you end up working for Coca Cola). Both the volume and surface area are determined by the diameter \( (d) \) of the cylinder and its height \( (h) \). The exact formulas for the surface area and volume are trivial and not really important here. Each solution to the problem is represented by the two values \((d,h)\) which define a cylinder of diameter \( d \) and height \( h \) – the cylinder is a feasible solution if the volume is \( \geq k \) ... and the solution is optimal if the surface area of the cylinder is as small as possible.

In the traditional genetic algorithm we would represent each \((d,h)\) pair as a vector of bits, and perform our cross-over and mutation operations on those vectors.

In a **real-valued genetic algorithm** we simply represent each solution by the pair of real numbers \((d,h)\)

But now we have to redefine mutation and cross-over.

Mutation is the easy one – we want each mutation to be a small change in the solution. One simple way to do this is to randomly pick one of the values and change it by a small percentage. The percentage change can be fixed or randomly chosen.
Cross-over is more difficult. If we start with a collection of \((d,h)\) pairs and just swap the elements, we probably won’t have much success. For example, if we use parents \((d_1', h_1')\) and \((d_2', h_2')\) to create offspring \((d_1, h_1)\) and \((d_2, h_1)\) we are pretty much limited to the \(d\) and \(h\) values in the initial population (with some changes due to mutation). We would need a huge initial population to get a good coverage of the solution space.

The most popular solution to this is to use what we call **arithmetic cross-over**.

Let’s suppose that we have two parent solutions, each defined by a vector of reals:

\[
P1 = [P1_1, P1_2, \ldots, P1_n] \quad \text{and} \quad P2 = [P2_1, P2_2, \ldots, P2_n]
\]

We can use our standard methods to decide which elements of the vectors will be involved in the cross-over:

- choose a value of \(k\) and cross-over to the right of that point
- choose \(k_1\) and \(k_2\) and cross-over the segment between those points
- choose some subset of \(P1\) and \(P2\) and cross-over those chosen elements

For simplicity I’m going to describe **complete arithmetic cross-over**, in which all elements of the vectors are involved in the cross-over.

We randomly choose a value \(\alpha\) in the open interval \((0..1)\), and compute the offspring \(C1\) and \(C2\) with

\[
C1_i = \alpha \cdot P1_i + (1 - \alpha) \cdot P2_i \\
C2_i = (1 - \alpha) \cdot P1_i + \alpha \cdot P2_i
\]

It’s easy to see that each of the offspring is a weighted average of the two parents – one with the weight shifted towards \(P1\) and the other shifted towards \(P2\). Note that if \(\alpha = 0.5\) then \(C1\) and \(C2\) are identical.

The problem with this is that it never produces a value outside the range of the values in the parents. Suppose we have a problem where the optimal solution (which we are trying to find) has a value of 20 for one of the elements. If all members of the initial population have values < 20 for that element, then the only way we will get to the optimal solution is by a lucky sequence of mutations.
A solution to this problem suggested in class is to let $\alpha$ take on values outside the interval (0..1). After working on this for a while last night I can’t see anything critically wrong with it – for large values of $\alpha$ the offspring may not be feasible solutions (for example, some of the elements of the offspring vectors might be outside of valid ranges) but we need to check offspring for viability anyway so this is not a problem.

However this solution does not seem to be discussed in the genetic algorithm literature that I have read – admittedly a tiny fraction of what is available. What follows is a method that is frequently mentioned: **simulated binary cross-over**

This method is described differently in different sources, but the gist is always the same: to produce offspring $C_1$ and $C_2$ from parents $P_1$ and $P_2$, we

1. choose a positive scale factor $\beta$
2. compute the absolute difference $d_i$ between each pair of elements involved in the cross-over
3. compute the mean $x_i$ of each pair of elements involved in the cross-over
4. compute the $i^{th}$ element of $C_1$ as $m_i - \beta \cdot d_i$
5. compute the $i^{th}$ element of $C_2$ as $m_i + \beta \cdot d_i$

i.e. $x_i = \frac{1}{2}(P_{1i} + P_{2i})$

$C_{1i} = x_i - \beta \cdot |P_{1i} - P_{2i}|$

$C_{2i} = x_i + \beta \cdot |P_{1i} - P_{2i}|$

If $\beta < \frac{1}{2}$, the values $C_{1i}$ and $C_{2i}$ will fall between $P_{1i}$ and $P_{2i}$

If $\beta > \frac{1}{2}$, the values $C_{1i}$ and $C_{2i}$ will fall outside $P_{1i}$ and $P_{2i}$

By randomly choosing the value of $\beta$ for each pairing we can produce offspring throughout the solution space.
One last example of cross-over mechanics:

Genetic algorithms have been applied to the Travelling Salesperson problem – one of our 8 problems from the start of the course. But none of the cross-over over techniques we have discussed so far are applicable.

Each solution consists of a permutation of the cities on the map. For simplicity we start every permutation at 1 (since each solution is a cycle, each solution can be rotated until 1 is in the first position), so each solution looks like (1, ....) where the .... is a permutation of \{2,3,...,n\}

But we can see immediately that our standard cross-over methods won’t work. If we work on the bit-vector and swap bits we are going to end up with things that aren’t permutations – and the same will happen if we try the arithmetic or simulated binary cross-overs.

One popular solution is based on the segment cross-over method: given two parent permutations P1 and P2, we choose a segment starting in position \(k_1\) and ending in position \(k_2\). Offspring C1 copies that segment from P1, and fills in the rest of the permutation by copying values from P2, maintaining their order from P2. Offspring C2 copies the chosen segment from P2 and fills in the rest of the permutation by copying values from P1, maintaining their order from P1.
An example will make this clear:

I have omitted most of the arrows from P1 to C2 to keep the figure easy to understand ... please feel free to add them as an exercise!

The main point of this example is that genetic algorithms can be adapted to a wide variety of problems through creative design of the cross-over operation.

Other applications that we didn’t mention include designing sorting networks, designing strategies for the Prisoner’s Dilemma, design radio antennae, and so on. Genetic algorithms provide a very powerful tool for finding good solutions to problems where finding the optimal solutions is computationally infeasible.

THE END