Genetic Algorithms

We have seen that Branch and Bound algorithms can be used to find optimal solutions to optimization problems (that is, problems where there are many solutions and the goal is to find a solution that either maximizes or minimizes (depending on the problem) the value of some function that is based on the details of the particular solution). For example, an instance of the Travelling Salesperson problem has \( \frac{(n - 1)!}{2} \) possible solutions – the goal is to choose one that has the minimum cost.

We have also seen that because they may have to examine many partial solutions, B&B algorithms often take exponential time in the worst case. On the good side, B&B algorithms are guaranteed to find an optimal solution.

We have another way of dealing with such optimization problems: find algorithms that run quickly, but which may not find the optimal solution.

Sometimes we can put a limit on how far off such an algorithm can be with respect to the optimal solution. For example, if the edge weights in an instance of the Travelling Salesperson Problem obey the triangle inequality then there is a fast algorithm that is guaranteed to find a solution which costs no more than twice the optimal cost. Similarly for a problem called Bin Packing\(^1\), a simple greedy algorithm is guaranteed to use no more than \( 1 + \frac{11}{9} \cdot \text{optimal} \) bins.

There are many other optimization problems for which we can prove that no such algorithm can exist unless \( P = NP \). Since most computer scientists (at least the ones who care) are confident that \( P \neq NP \) we are fairly certain that these hypothetical fast approximation algorithms do not exist.

---

\(^1\) Given a set of \( n \) objects, each with known mass, and an infinite supply of bins, each of capacity \( k \) kilograms, what is the minimum number of bins needed to contain all the objects, with each bin containing \( \leq k \) kg of total mass?
For these problems we can try yet another approach: create an algorithm that usually runs fairly quickly, and usually finds a fairly good solution.

The fundamental principle of Branch & Bound algorithms is to search for an optimal solution by building partial solutions up step by step until they are complete. In the algorithms we will consider now, the fundamental principle is to start with one or more complete solutions, and then repeatedly improve them until we reach an optimal solution (or at least a good solution).

A very simple example of this approach is called hill-climbing. Suppose we have a function \( f(x) \) about which we know nothing, but for any \( x \) we can compute \( f(x) \). Our goal is to find a value of \( x \) that maximizes \( f(x) \). The hill-climbing approach looks something like this:

```
x = a randomly chosen number
step = 1
min_step = 0.001
threshold = 0.01
diff = threshold + 1
while diff > threshold and step > min_step:
    xm = x - step
    xp = x + step
    if f(xm) > f(xp):
        next = xm
    else:
        next = xp
    if f(next) < f(x):
        step = step/2
    else:
        diff = abs(next - x)
x = next
```

The basic idea is to look at the “neighbours” of \( x \) on each side, and move to the one that increases the value of \( f(x) \) the most (that is, we try to move “up the hill”). If both directions go down, we decrease the step size and see if that lets us improve the value of \( f(x) \) a bit more by moving a smaller amount. We exit the loop either when the step size is below the minimum step limit we have set, or when there is no noticeable improvement in the value of \( f(x) \).

This is a very “bare-bones” version but it shows the basic idea, and it also reveals one of the major problems with hill-climbing algorithms: the algorithm may terminate because it has reached a local maximum, rather than a global maximum (optimal) solution.
Hill-climbing may climb the wrong hill! There are ways to improve the quality of the result. One is to run the hill-climbing algorithm many times with different random starting points. With luck one of them will be on the “slopes” of the tallest hill. Another is to let the algorithm run until it reaches a maximum value, then jump to a new $x$ quite a distance away (this is an intentionally vague term) and see if it climbs to a different maximal solution. We can also try to reach a maximum faster by taking big steps when the function is increasing slowly.

Another, more daunting problem with hill-climbing algorithms is revealed when we are dealing with more than one variable. For example, suppose our problem involves assigning each of $m$ chocolate bars to one of $n$ students (so each solution consists of a vector of length $m$, with each value in the range $[1..n]$). If we consider a “neighbour” of a solution to be the result of changing each variable by -1, 0 or 1, then the total number of neighbours for most solutions is $3^m - 1$ ... unless $m$ is small this makes each iteration of the hill-climbing algorithm very time consuming.

All of which brings us at last to the topic of genetic algorithms. Genetic algorithms were invented in the 1960s by Michael Holland as a way to model biological inheritance of genetic information, but it was quickly realized that they offer a novel and often very effective alternative to hill-climbing and other heuristic search algorithms.

The modelling concept goes like this: in a population of biological members of a species, some members are healthier than others. The healthier specimens are more likely to engage in mating, and thus their genetic information is more likely to propagate into the next generation. When members mate, their offspring inherit genetic information from both parents. Some of the offspring are subject to random mutation in their genetic material.
Finally, since resources are finite, the environment can only support a limited population.

In terms of our application of this model to problem-solving, the population consists of solutions to the problem (these solutions are whimsically called *chromosomes* in the literature). Each solution is typically represented by a vector of bits or numeric values. The health of each individual is its value as computed by the objective function. The relative health values of the individuals are used to determine which of them are more likely to reproduce.

When two individuals reproduce, they create two offspring. Each offspring has part of its vector from one parent, and the other part from the other parent. Each offspring may or may not undergo a mutation.

Once all new offspring have been created, the population is reduced to be within the limit imposed by the environment.

This process repeats until the maximum health within the population is no longer increasing.

Some of the potential benefits of this approach should already be apparent:

- by starting with a large initial population, the chances are increased that some individual will be “close” to the optimal solution
- the mating selection process builds on the strength of the best solutions found so far, while not ignoring the others
- by combining the information from individuals that are very different, new solutions can be generated that are far from either parent
- the algorithm never computes all the neighbours of any individual, so each new generation can be created quickly even if each individual is represented by a long vector of values
- the population reduction phase keeps the memory requirement low.

Over the next few days we will explore each step of this process more carefully, and apply it to specific problems.