Advanced Modelling in Biology

Question 5

Define combinatorial optimization. Give one example of such a problem and explain what makes it usually a computationally hard problem.

Combinatorial optimization is the minimization or maximization of functions with discrete output sets. One example of such a problem is the travelling salesman problem where one must find the minimum distance to travel through N number of cities. Although there are algorithms (e.g. Dijkstra's algorithm) that generally yield the solution, most of these problems are only solved by complete enumeration (ie going through every possible solution).

If we look closely at the travelling salesman problem with N number of cities and each city connected to the each other, there are N! number of possible routes the salesman can take. Because computation then also becomes on the order of N! (known as combinatorial explosion), these types of problems are generally computational hard.

Two methods used in combinatorial optimization are evolutionary (genetic) algorithms and simulated annealing. Describe the origin and basic ideas behind these algorithms and compare their advantages and disadvantages.

The origin of simulated annealing comes from materials science and is based upon the idea of exploring the energy function by jumping to different energies depending upon the "temperature". One first selects an initial condition can calculates the energy at that point. Another point close by is randomly selected, and if the energy at that point is less than the previous energy, then we move to the lower energy point. However, with some probability related to the Boltzmann distribution, we also accept increases in energy changes. Higher temperatures lead us to have a probability of accepting higher temperature increases and tends to ensure that we do not get stuck at a local minima. We repeat the algorithm, lowering the temperature each time, until we are satisfied that we have reached the global minimum.

Evolutionary (or genetic) algorithms on the other hand were motivated from biology and is based upon the idea of parallel exploration of state space, leading to faster convergence of solutions. Given a discrete set of solutions to our problem, we initially pick a population P of solutions at random. The population first undergoes reproduction where pairs of solutions create one offspring. Each solution in the population undergoes mutation at a certain frequency as well as cross-over (chiasma) where portions of solutions are chopped off and transferred to another solution. The population is then ranked based upon the cost function and the bottom P/2 are removed from the population eliminating poor solutions. The algorithm again is repeated until a suitable solution is reached.

For continuous functions, it is easy to see that simulated annealing would be more appropriate than genetic algorithms. However, due to the parallel exploration of state space utilized by genetic algorithms, they converge much faster to a solution than simulated annealing. Furthermore, genetic algorithms are able to explore the state space much more through mutations and crossovers than the energy space simulated annealing covers. One disadvantage of simulated annealing that it usually gets caught in local minima and depending on the shape of the energy landscape, it may be difficult to exit the local minima to continue the energy landscape exploration. This is overcome by having a very slow decrease in temperature and allowing it to run for longer. It is important to note that both of these

algorithms are heuristic and do not guarantee that the solution be found. Usually, they are run several times to test convergence onto a particular solution. Only when run an infinite number of times do the algorithms guarantee the optimal solution.

For both algorithms, write a few lines of pseudo-code to apply them to the optimization of a function U(x), where x is a configuration of the discrete set X = {x}.

For simulated annealing

```
Pick a random configuration x
Evaluate U(x)
For 1:number of times to repeat {
        Generate a new configuration xnew by picking a random element in x and changing
        it's state
        Evaluate U(xnew)
        If U(xnew) < U(x)
                Accept change in configuration, U(x) = U(xnew)
        Else if rand() < e^-(U(x)-U(xnew))/kT)
                Accept change in configuration, U(x) = U(xnew)
        Else:
                Do not accept change
        Decrease temperature T
        }
```

For genetic algorithms

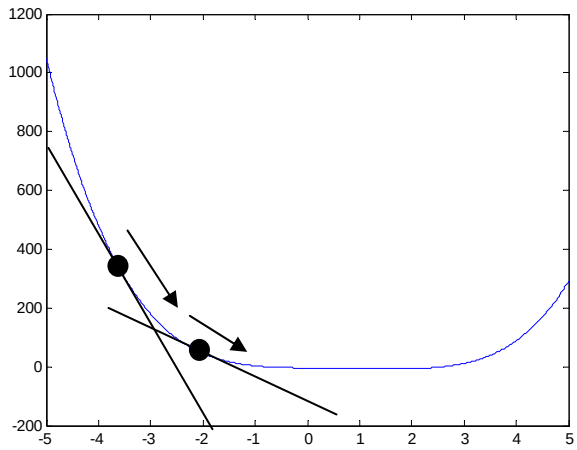
```
Generate P number of configurations x
Evaluate U(x) for all solutions in P
For 1:number of times to repeat {
        Pair up random configurations
        Mate the pairs of configurations to produce children
        Mutate each configuration with a probability p
        Take pairs of random configurations and perform crossover
        Evaluate U(xnew) for all solutions in expanded population
        Rank all solutions depending upon U
        Kill off bottom P/2 solutions
        }
```

Question 6

Use a sketch to explain the mathematical basis of gradient methods for the optimization of a function $U(x_1, x_2, \ldots, x_n)$.

The mathematical basis of gradient methods depends on calculating the slope (or gradient for dimensions greater than 2) and travelling along that pathway to reach the minimum. For gradient methods however, we are only guaranteed a local optimization and not global optimization. Only for convex functions are we guaranteed a global minimum. For normal gradient methods such as steepest descent, an initial starting position is taken randomly and the gradient is calculated at that point. Steps are taken against the direction of the gradient

$-\nabla U$ until the gradient at the point becomes zero signifying that you have reached a minimum. This is seen in the diagram below.



You need to obtain numerically the global minimum of the function U(x) in Figure 1 for which you have the explicit analytical form. Compare in detail how you would achieve it:
i) *by using an integrator such as ode45*
ii) *by using the steepest descent solver fsolve.*

Both ode45 and fsolve by themselves will only achieve finding the local minima. With ode45, we integrate the negative energy gradient, such that we are always progressing down the energy gradient. Once it reaches the minima, ode45 will then stop. So given our energy function U(x), we can calculate ∇U and hence come up with our differential equation to plug into ode45 as $\frac{dx}{dt} = -\nabla U$. Over time, *x* will converge to a local minimum. To find the global minimum of the function, we must try several different initial conditions for *x* and evaluate the energy U(x) for each minimum that is found using the ode45. Only by trying a sufficient number of initial conditions that yield all of the local minima can we be sure that we are indeed at a global minimum.

Fsolve works in a similar way to the above method and is also an implementation of the steepest descent algorithm. However, fsolve finds the zeros of a function and so the input function must be the derivative of our energy function U(x) that we wish to minimize. Again, with one initial condition, you are guaranteed a local minimum, but in order to find the global minimum, several initial conditions must be tried and the energy U(x) must be evaluated at each zero to ensure that we really reach the global minimum.

For both methods, discrete time steps are not generally taken unless specified (when using ode45). Instead, time steps are taken to maximize the efficiency of the algorithm so it quickly converges to a solution in matlab.

Use this example to discuss the problems encountered when applying gradient methods to global optimization of non-convex functions. Explain why this is a computationally hard problem in general and mention briefly some of the methods that can help with it.

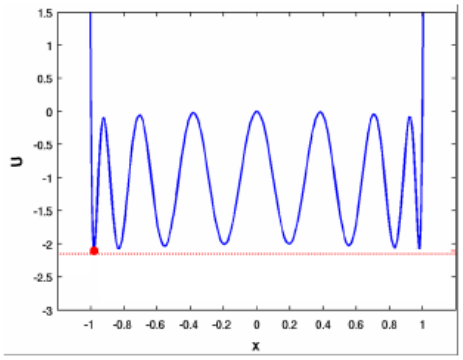


Figure 1:

One of the main problems with gradient methods is that you are not guaranteed a solution. As is seen in figure 1, if we start at different initial conditions, we may or may not end up in the global minimum but might get stuck in a local minimum. This problem becomes exacerbated the more minima we have. In the worst case scenario, one must try an infinite number of initial conditions to ensure that the global minimum is found, making this a computationally difficult problem. One can get around this by simulated annealing where an initial guess is selected at random, and a step size is taken. Although we accept all changes that lead us down a gradient, we will also accept changes that take us up a gradient with some probability related to the "temperature" that we set and the change in energy that the move causes. This also does not guarantee the global minima, but by allowing movement up the gradient, we are less likely to get stuck in a local minimum. Only if we allow the temperature to decrease very slowly and run the algorithm for an infinite amount of time will we be guaranteed the global minimum.

Another problem with gradient methods is quite the opposite to having several minima. Having a very shallow function such that the gradient changes very little for a long time can also cause the computer to take a long time to converge. Both fsolve and ode45 tend to take time steps that will maximize the change in gradient, but for shallow functions, the change in gradient might be too small to converge quickly and will have to keep taking infinitely small time steps until the minimum is reached. For energy landscapes of more than 2 variables, conjugate gradient methods can be applied where we don't always take steps in the direction against the gradient, but move to the side and then return back to the gradient.