Stochastic Search

A field that draws heavily on the development of efficient algorithms is artificial intelligence (AI). Herb Simon once asserted that all of AI can be reduced to search. AI is also a heavy user of randomized methods, so it seems natural to conclude our discussion of randomized algorithms by considering techniques for randomization in search.

One approach to search that is used heavily in optimization and artificial intelligence is called "iterative improvement search" (or sometimes "local search"). Consider a problem you are attempting to solve where you begin by constructing a complete "candidate" solution. It is not expected that this candidate solution will actually solve the problem, but it is going to be used as a starting point for searching for the solution. Now consider some operations you can apply to the candidate solution to change it to another candidate. Then search will reduce to generating these sequences of candidate solutions until the desired solution is found.

As we see, this approach to search can be considered a form of graph search. Specifically, we can associate a vertex in the graph with each of the candidate solutions and then associate an edge to an operator that transforms one candidate to another. Next we can apply a "fitness" value to each of the states to estimate how far we are from finding the desired solution. Search then amounts to optimizing the fitness.

Hill Climbing

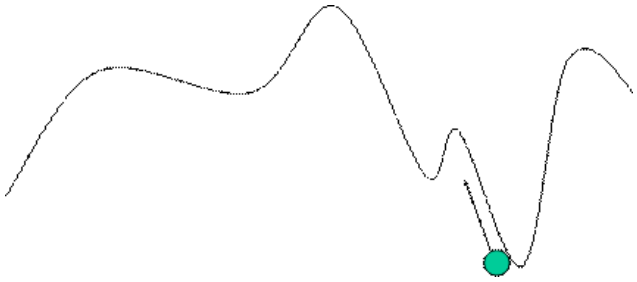
As described, one way to focus our search is to consider the fitness of the adjacent candidate solutions and pick the one that improves the fitness the most. This is called hill climbing, and the following pseudo-code formalizes the approach.

Algorithm 1 Hill Climbing

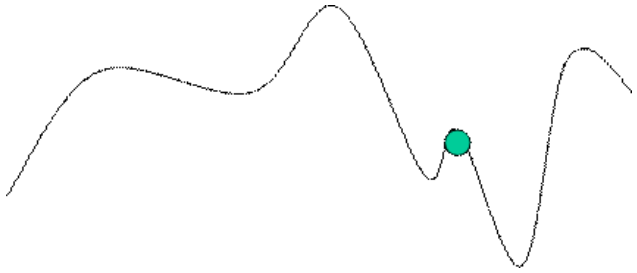
```
HillClimb(problem)
// Maximization problem
  input: problem
  output: curr, next
  curr ← MakeNode(InitState(problem))
  for t ← 1 to ∞ do
    next ← max(neighbor(curr))
    if val(next) > val(curr) then
      curr ← next
    else
      return curr
  end do
```

As shown, the hill climbing algorithm is represented as the graph traversal algorithm described above. The neighbors of node are defined to be the candidate solutions reachable from the current candidate as a result of applying only one operator. This code shows a maximization problem where we accept the highest-valued neighboring node as long as it is better than the current node. This can easily be converted to a minimization problem by reversing the direction of the inequality.

Notionally, we can imagine this fitness evaluation function defining a landscape that we are searching. For example, the following shows a candidate solution at the base of a hill about to climb.



If the search algorithm explores the landscape as described, it will end up at the peak to its left.



While this approach is widely used, it suffers from some serious drawbacks. First (and foremost), there is no guarantee that the algorithm will find the absolute best solution in the space. This is because the localized way candidate solutions are explored could result in the search algorithm getting "stuck" in a local optimum (as shown here). Second, this approach also depends on being able to construct candidate solutions. This in itself may require some amount of search or some other algorithm.

Multiple Restarts

For the first "randomized" search algorithm, we are going to modify the hill climbing algorithm in a straightforward way. Specifically, all we will do is run multiple hill climbs from different, randomly generated, starting positions and select the solution that is the best. Does this guarantee that we will find the optimal solution? In the limit, yes. Practically speaking, no. Let's see if we can find a better way.

Simulated Annealing

Simulated annealing also builds upon the basic hill climbing approach but uses a slightly more complicated approach to randomizing. What is interesting is that, with a properly constructed neighborhood function, simulated annealing also guarantees convergence to a global optimum. In theory, this convergence is still "in the limit," but considerable empirical evidence shows it finding very good solutions in a reasonable amount of time.

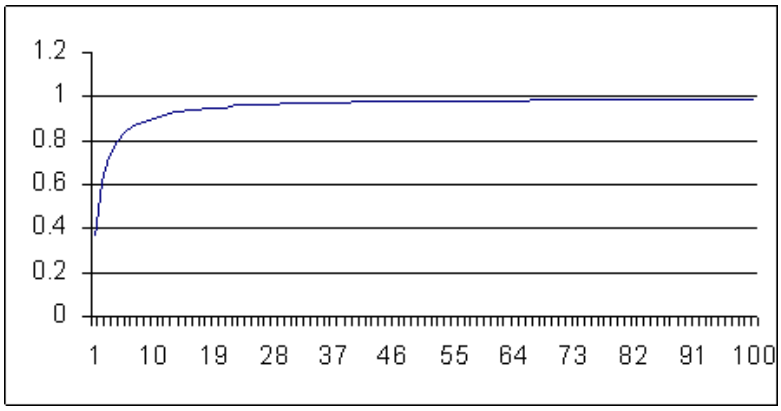
The basic idea in simulated annealing is that we are going to modify the action taken when the selected neighboring candidate is not better than the current candidate. In simple hill climbing, the algorithm would terminate. In simulated annealing, we will accept the degraded candidate according to the Boltzman probability distribution. The following provides a pseudo-code representation of the algorithm.

Algorithm 2 Simulated Annealing

```
SimulatedAnnealing(problem, schedule)
// Maximization problem
  input: problem, schedule
  output: curr, next, T
  curr ← MakeNode(InitState(problem))
  for t ← 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← rand(neighbor(curr))
    ΔE ← val(next) − val(curr)
    if ΔE > 0 then
      curr ← next
    else
      curr ← next with probability exp(ΔE/kT
  end do
```

This time, we do not consider all of the neighbors of the candidate solution but select one of them at random. Then we calculate its fitness and compare to the current candidate. If the fitness of the neighbor is better, then like hill climbing, we move to that candidate. On the other hand, if the candidate is worse, we calculate a probability of accepting it anyway based on how much worse it is (ΔE).

The randomness is also driven by the parameter T. One can think of this parameter as "temperature" that is reduced gradually over time. This is where the name "simulated annealing" comes from since temperature is reduced gradually when annealing materials to make them stronger. With this temperature parameter, we see that when temperature is high, the probability of accepting a degraded candidate is also high. As the temperature decreases, so does the probability, until it reaches hill climbing in the limit. This can be shown graphically as follows:



$$y = e^{\Delta E/kT}$$

$$\Delta E < 0$$

The parameter k is user defined and is used to "shape" the distribution as desired.

Key to this process (in addition to the definition of the neighborhood function) is the annealing schedule. If one anneals to fast, then hill climbing takes over too soon. If one anneals too slowly, then a lot of wasted searching results. Some common criteria for defining an appropriate annealing schedule are as follows:

$$\sum_{t=1}^{\infty} T_t = \infty$$

$$\sum_{t=1}^{\infty} T_t^2 < \infty.$$

One schedule that meets these criteria is

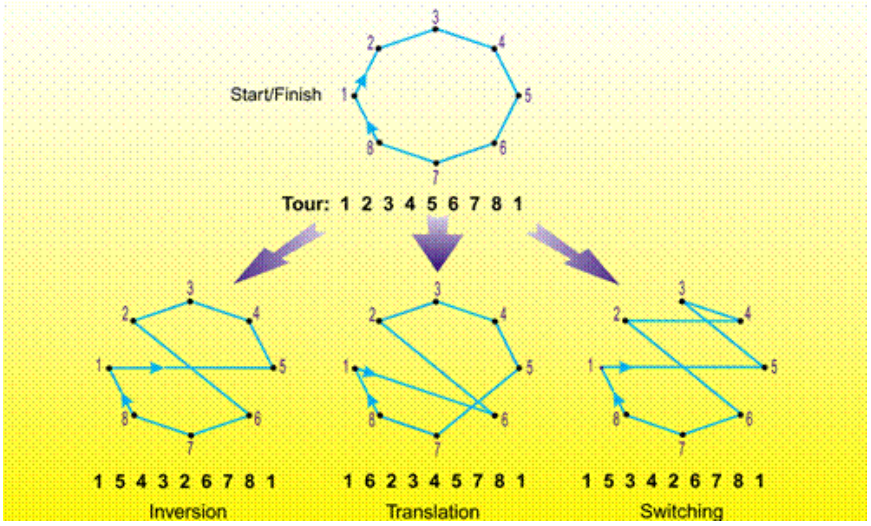
$$T_t = \frac{T_0 \tau}{\tau + n_t}$$

where T_0 is the initial temperature, n_t is the number of temperature updates, and τ is a user defined parameter to control the rate.

Solving TSP

As an example, consider the Traveling Salesperson Problem (TSP) again. Recall that we proved that TSP is NP-complete. It is interesting that simulated annealing has been applied many times to TSP with considerable success. To apply simulated annealing, we start by generating a candidate tour (which can be any legal tour) and then apply modification operators to the tour to generate new candidate tours. We will use three different operators—inversion, translation, and switching.

With inversion, we take a subsequence in the tour and reverse the direction. With translation, we take a subsequence of the tour and shift it between two different consecutive cities. Finally, with switching, we take two cities and swap their locations in the tour. These operations are illustrated as follows:



In this example, inversion reverses the subsequence 2–3–4–5, Translation moves the subsequence 2–3–4–5 between cities 6 and 7, and switching swaps cities 2 and 5.