Hill Climbing

In numerical analysis, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will likely be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing finds optimal solutions for convex problems - for other problems it will find only local optima (solutions that cannot be improved upon by any neighbouring configurations), which are not necessarily the best possible solution (the global optimum) out of all possible solutions (the search space). Examples of algorithms that solve convex problems by hill-climbing include the simplex algorithm for linear programming and binary search. To attempt to avoid getting stuck in local optima, one could use restarts (i.e. repeated local search), or more complex schemes based on iterations (like iterated local search), or on memory (like reactive search optimization and tabs search), or on memory-less stochastic modifications (like simulated annealing).

Pseudocode:`

Algorithm Discrete Space Hill Climbing	**Algorithm Continuous Space Hill Climbing is**

currentNode := startNode
loop do
L := NEIGHBORS(currentNode)
nextEval := −INF
nextNode := NULL
for all x in L do
if EVAL(x) > nextEval then
nextNode := x
nextEval := EVAL(x)
if nextEval ≤ EVAL(currentNode) then
// Return current node since no better neighbors exist
return currentNode
currentNode := nextNode
	currentPoint := initialPoint // the zero-magnitude vector is common
	stepSize := initialStepSizes // a vector of all 1's is common
	acceleration := someAcceleration // a value such as 1.2 is common
	candidate[0] := −acceleration
	candidate[1] := −1 / acceleration
	candidate[2] := 1 / acceleration
	candidate[3] := acceleration
	bestScore := EVAL(currentPoint)
	loop do
	beforeScore := bestScore
	for each element i in currentPoint do
	beforePoint := currentPoint[i]
	bestStep := 0
	for j from 0 to 3 do // try each of 4 candidate locations
	step := stepSize[i] × candidate[j]
	currentPoint[i] := beforePoint + step
	score := EVAL(currentPoint)
	if score > bestScore then
	bestScore := score
	bestStep := step
	if bestStep is 0 then
	currentPoint[i] := beforePoint
	stepSize[i] := stepSize[i] / acceleration
	else
	currentPoint[i] := beforePoint + bestStep
	stepSize[i] := bestStep // accelerate
	if (bestScore − beforeScore) < epsilon then
	return currentPoint