Hill Climbing

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, and then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the traveling salesman problem. It is easy to find an initial solution that visits all the cities but will 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 is good for finding a local optimum (a solution that cannot be improved by considering a neighboring configuration) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization and tabu search, on memory-less stochastic modifications, like simulated annealing.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends.

Mathematical description

Hill climbing attempts to maximize (or minimize) a target function f(x), where x is a vector of continuous and/or discrete values. At each iteration, hill climbing will adjust a single element in x and determine whether the change improves the value of f(x). (Note that this differs from gradient descent methods, which adjust all of the values in x at each iteration according to the gradient of the hill.) With hill climbing, any change that improves f(x) is accepted, and the process continues until no change can be found to improve the value of f(x). x is then said to be "locally optimal".

In discrete vector spaces, each possible value for x may be visualized as a vertex in a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f(x), until a local maximum (or local minimum) *xm* is reached.

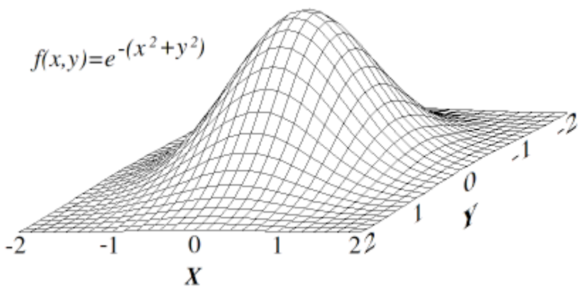
Variants

In simple hill climbing, the first closer node is chosen, whereas in steepest ascent hill climbing all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space, which are not solutions. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.

Stochastic hill climbing does not examine all neighbors before deciding how to move. Rather, it selects a neighbor at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

Random-restart hill climbing is a meta-algorithm built on top of the hill-climbing algorithm. It is also known as Shotgun hill climbing. It iteratively does hill-climbing, each time with a random initial condition x_0. The best *xm* is kept: if a new run of hill climbing produces a better *xm* than the stored state, it replaces the stored state.

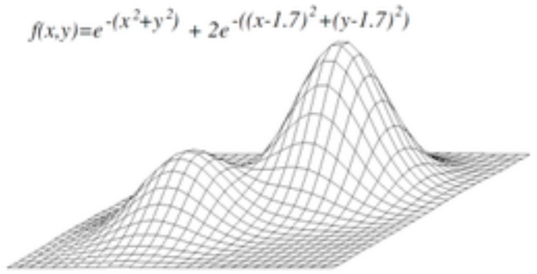
Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, than carefully optimizing from an initial condition.



A visualization of a convex surface. Hill-climbers are well suited for optimizing over such surfaces, and will converge to the global maximum.

Problems

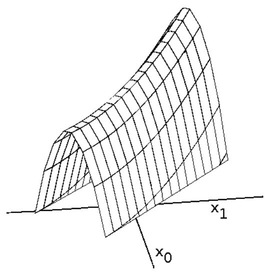
Local maxima



A visualization of a surface with two local maxima. (Only one of them is the global maximum.) If a hill-climber begins in a poor location, it may converge to the lower maximum.

A problem with hill climbing is that it will find only local maxima. Unless the heuristic is convex, it may not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing.

Ridges and Alleys

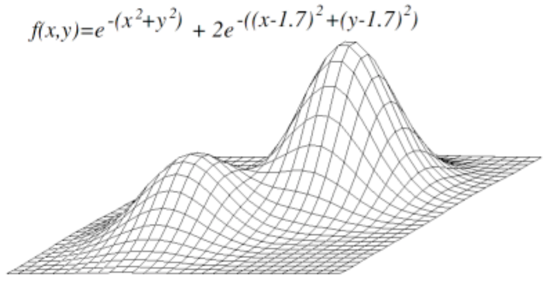


A visualization of a ridge

Ridges are a challenging problem for hill climbers that optimize in continuous spaces. Because hill climbers only adjust one element in the vector at a time, each step will move in an axis-aligned direction. If the target function creates a narrow ridge that ascends in a non-axis-aligned direction (or if the goal is to minimize, a narrow alley that descends in a non-axis-aligned direction), then the hill climber can only ascend the ridge (or descend the alley) by zig-zagging. If the sides of the ridge (or alley) are very steep, then the hill climber may be forced to take very tiny steps as it zig-zags toward a better position. Thus, it may take an unreasonable length of time for it to ascend the ridge (or descend the alley).
By contrast, gradient descent methods can move in any direction that the ridge or alley may ascend or descend. Hence, gradient descent is generally preferred over hill climbing when the target function is

differentiable. Hill climbers, however, have the advantage of not requiring the target function to be differentiable, so hill climbers may be preferred when the target function is complex.

Plateau
Another problem that sometimes occurs with hill climbing is that of a plateau. A plateau is encountered when the search space is flat, or sufficiently flat that the value returned by the target function is indistinguishable from the value returned for nearby regions due to the precision used by the machine to represent its value. In such cases, the hill climber may not be able to determine in which direction it should step, and may wander in a direction that never leads to improvement.



A visualization of a surface with two local maxima. (Only one of them is the global maximum.) If a hill-climber begins in a poor location, it may converge to the lower maximum.

Pseudocode

```
Discrete Space Hill Climbing Algorithm
   currentNode = startNode;
   loop do
      L = NEIGHBORS(currentNode);
      nextEval = -INF;
      nextNode = NULL;
      for all x in L
         if (EVAL(x) > nextEval)
              nextNode = x;
              nextEval = EVAL(x);
      if nextEval <= EVAL(currentNode)
         //Return current node since no better neighbors exist
         return currentNode;
      currentNode = nextNode;
Continuous Space Hill Climbing Algorithm
   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] = 0;
   candidate[3] = 1 / acceleration;
   candidate[4] = acceleration;
   loop do
      before = EVAL(currentPoint);
      for each element i in currentPoint do
         best = -1;
         bestScore = -INF;
         for j from 0 to 4         // try each of 5 candidate locations
            currentPoint[i] = currentPoint[i] + stepSize[i] * candidate[j];
            temp = EVAL(currentPoint);
            currentPoint[i] = currentPoint[i] - stepSize[i] * candidate[j];
            if(temp > bestScore)
                 bestScore = temp;
                 best = j;
         if candidate[best] is not 0
            currentPoint[i] = currentPoint[i] + stepSize[i] * candidate[best];
            stepSize[i] = stepSize[i] * candidate[best]; // accelerate
      if (EVAL(currentPoint) - before) < epsilon
         return currentPoint;
```

References

Russell, Stuart J.; Norvig, Peter (2003), *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 111–114, ISBN 0-13-790395-2