Local Search-Single State method Hill-Climbing

Lecture#2

Esmaeil Nourani

Local Serach

you need to be able to do four things when you want to optimize an ill-structure problem:

- Provide one or more initial candidate solutions. This is known as the **initialization procedure.**
- Assess the Quality of a candidate solution. This is known as the **assessment procedure.**
- Make a Copy of a candidate solution.
- Tweak a candidate solution, which produces a *randomly slightly different candidate solution.*

  This, plus the Copy operation are collectively known as the **modification procedure.**

Hill Climbing

This technique is related to gradient ascent, but it doesn't require you to know the strength of the gradient or even its direction: you just iteratively test new candidate solutions in the region of your current candidate, and adopt the new ones if they're better. This enables you to climb up the hill until you reach local optima.

Gradient Ascent

```
Algorithm *Gradient Ascent*
1: $x \leftarrow$ random initial vector
2: repeat
3:     $x \leftarrow x + a * gradient(f(x))$
4: until $x$ is the ideal solution or we have run out of time
5: return $x$
```

Hill-Climbing

```
Algorithm  Hill-Climbing
1: $S \leftarrow$ some initial candidate solution     Initialization Procedure
2: repeat
3:       $R \leftarrow$ Tweak(Copy($S$))                Modification Procedure
4:        if Quality($R$) > Quality($S$) then
                                      Assessment and Selection Procedures
5:                  $S \leftarrow R$
6: until $S$ is the ideal solution or we have run out of time
7: return $S$
```

Candidate Solution Representation

The initialization, Copy, Tweak, and fitness assessment functions collectively define the **representation of your candidate solution. Together they show how your candidate** solution is made up and how it operates.

If you can create the four functions above in a reasonable fashion, you're in business.

Candidate Solution

What might a candidate solution look like?
It could be:
a vector; or
an arbitrary-length list of objects; or
an unordered set or
collection of objects; or
a tree; or a graph. Or
any combination of these.
Whatever seems to be appropriate to your problem.

Candidate Solution

- One simple and common representation for candidate solutions is the same as the one used in the gradient methods:

  a **fixed-length vector of real-valued numbers.**

  Creating a random such vector is easy: just pick random numbers within your chosen bounds. If the bounds are *min and max inclusive, and the vector length is l.*

Random Real-Value Generator

```
Algorithm Generate a Random Real-Valued Vector
1: min ← minimum desired vector element value
2: max ← maximum desired vector element value
3: v ← a new vector (v1, v2, ...vl)
4: for i from 1 to l do
5:     vi ← random number chosen uniformly between min and max inclusive
6: return v
```

What is Tweak

To Tweak a vector we might (as one of many possibilities) add a small amount of random noise to each number: in keeping with our present definition of Tweak, let's assume **for now that** this noise is no larger than a small value *1.*

To Tweak a vector

```
1: $v$ ← vector $(v1, v2, ...vl)$ to be modified
2: $p$ ← probability of adding noise to an element in the vector " Often $p = 1$
3: $r$ ← half-range of uniform noise
4: $min$ ← minimum desired vector element value
5: $max$ ← maximum desired vector element value
6: for $i$ from 1 to $l$ do
7:     if $p \geq$ random number chosen uniformly from 0.0 to 1.0 then
8:         repeat
9:             $n$ ← random number chosen uniformly from $-r$ to $r$ inclusive
10:        until $min \leq vi + n \leq max$
11:        $vi$ ← $vi + n$
12: return $v$
```

The size of the bound of Tweak is r

Stochastic Procedure

- Notice the strong resemblance between Hill-Climbing and Gradient Ascent. The only real difference is that Hill-Climbing's more general Tweak operation must instead rely on a **stochastic** (partially random) approach to hunting around for better candidate solutions. Sometimes it finds worse ones nearby, sometimes it finds better ones.

Steepest Ascent Hill-Climbing

We can make this algorithm a little more aggressive: create *n “tweaks” to a candidate solution* all at one time, and then adopt the best one. This modified algorithm is called **Steepest Ascent Hill-Climbing**,

Steepest Ascent Hill-Climbing

- **Algorithm *Steepest Ascent Hill-Climbing***

```
1: n ← number of tweaks desired to sample the gradient
2: S ← some initial candidate solution
3: repeat
4:    R ← Tweak(Copy(S))
5:    for n − 1 times do
6:        W ← Tweak(Copy(S))
7:        if Quality(W) > Quality(R) then
8:            R ← W
9:    if Quality(R) > Quality(S) then
10:       S ← R
11: until S is the ideal solution or we have run out of time
12: return S
```

The Size of the bound of Tweak

- We know that *r is the size of the bound on Tweak.*

If the size is very small ➔

then Hill-Climbing will march right up a local hill and be unable to make the jump to the next hill because the bound is too small for it to jump that far. Once it's on the top of a hill, everywhere it jumps will be worse than where it is presently, so it stays put.

if the size is large ➔

then Hill-Climbing will reject around a lot. Importantly, when it is near the top of a hill, it will have a difficult time converging to the peak, as most of its moves will be so large as to overshoot the peak.

Exploration←→ Exploitation

- Notice how similar this is to *a(step) used in Gradient Ascent. This is one way of controlling* the degree of **Exploration versus Exploitation in our Hill-Climber.**

- **Optimization algorithms** which make largely local improvements are *exploiting the local gradient, and*
- ***Optimization algorithms*** *which mostly* wander about randomly are thought to *explore the space.*

The “uglier” the space, the more you will have no choice but to use a more explorative algorithm.

Single-State Global Optimization Algorithms

A **global optimization algorithm is one which, if we run it long enough, will *eventually find the*** global optimum.

The single-state algorithms we’ve seen so far cannot guarantee this.

Tweak may not be strong enough to get us out of it. Thus the algorithms so far have been **local optimization algorithms.**

Random Search

- Algorithm to find the global optimum ➔ simplest one → Random Search
- **Algorithm *Random Search***

```
1: Best ← some initial random candidate solution
2: repeat
3:     S ← a random candidate solution
4:     if Quality(S) > Quality(Best) then
5:         Best ← S
6: until Best is the ideal solution or we have run out of time
7: return Best
```

Random Search is the extreme in exploration (and global optimization); in contrast, Hill-Climbing , with Tweak set to just make very small changes and never make large ones, may be viewed as the extreme in exploitation (and local optimization)

Local/global

But there are ways to achieve reasonable exploitation and still have a global algorithm.

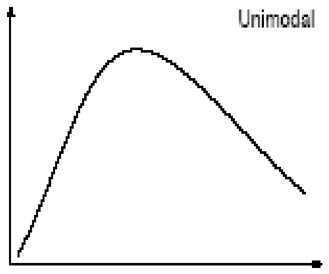
Iterative Hill-Climbing

Algorithm *Hill-Climbing with Random Restarts*

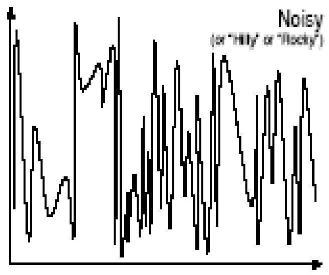
```
1:  T ← distribution of possible time intervals
2:  S ← some initial random candidate solution
3:  Best ← S
4:  repeat
5:      time ← random time in the near future, chosen from T
6:      repeat
7:          R ← Tweak(Copy(S))
8:          if Quality(R) > Quality(S) then
9:              S ← R
10:     until S is the ideal solution, or time is up, or we have run out of total time
11:     if Quality(S) > Quality(Best) then
12:         Best ← S
13:     S ← some random candidate solution
14: until S is the ideal solution or we have run out of total time
15: return Best
```

Numerical Test

Hill-Climbing is close to optimal, and where Random Search is a very bad pick.



Hill-Climbing is quite bad Random Search is expected to be very good



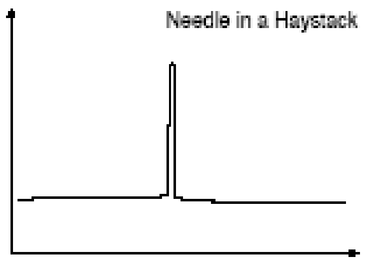
Smoothness criterion

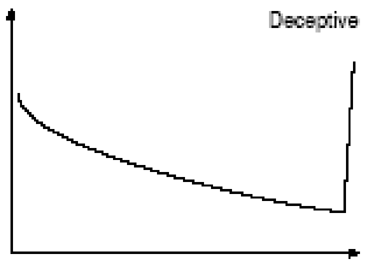
The difference is that in *Unimodal there is a strong relationship between the distance (along the x axis) of two* candidate solutions and their relationship in quality: similar solutions are generally similar in quality.

In the *Noisy situation, there's* no relationship like this: even similar solutions are very dissimilar in quality. This is often known as the **smoothness criterion for local search to be effective.**

Numerical Test

Random Search is the only real way to go, and Hill-Climbing is quite poor.
"It is pretty smooth"





Hill-Climbing not only will not easily find the optimum, but it is actively *let away from the optimum.*

Exploration versus Exploitation

you'd *like to use a highly exploitative algorithm (it's fastest), but*

the "uglier" the space, the more you will have no choice but to use a more explorative algorithm.

Global Optimization

- **Adjust the Modification Procedure**
- *Exploration vs. Exploitation The more large, random changes, the more exploration.*

- **• Adjust the Selection Procedure**
- *Exploration vs. Exploitation The more often you go down hills, the more exploration.*

- **• Jump to Something New**
- *Exploration vs. Exploitation The more frequently you restart, the more exploration.*

- **• Use a Large Sample Try many candidate solutions in parallel.**
- *Exploration vs. Exploitation More parallel candidate solutions, more exploration.*

Adjusting the Modification Procedure

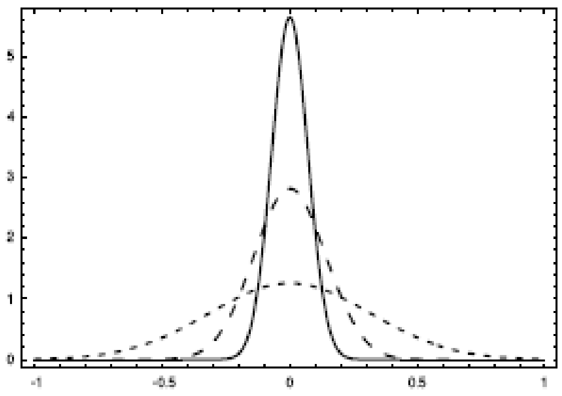


Figure 7 Three Normal or Gaussian distributions $N(\mu, \sigma^2)$ with the mean $\mu = 0$ and the variance σ^2 set to $\sigma^2 = 0.005$: ——, $\sigma^2 = 0.02$: – – –, and $\sigma^2 = 0.1$: - - - -.

Adjusting the Modification Procedure

```
Algorithm  Gaussian Convolution
1: !v ← vector (v1, v2, ...vl) to be convolved
2: p ← probability of adding noise to an element in the vector   " Often p = 1
3: σ² ← variance of Normal distribution to convolve        " Normal = Gaussian
4: min ← minimum desired vector element value
5: max ← maximum desired vector element value

6: for i from 1 to l do
7:         if p ≥ random number chosen uniformly
               from 0.0 to 1.0 then
8:                 repeat
9:                     n ← random number chosen from the NormaL
                           distribution N(0, σ² )
10:                until min ≤ vi + n ≤ max
11:                vi ← vi + n
12: return v
```

Ref

- Slides adapted from Advanced Algorithms course, presented by Dr. kourosh ziarati