Adam Mischke
Dr. Phillips
CSCI 4350 Open Lab 2
Tue. September 25, 2017

Solving Problems by Optimization

Problem:

Finding the maximum of a function can be hard, especially when one doesn't know what the function looks like (in terms of 2-d and 3-d), or if a function has a higher order of dimension that we aren't entirely familiar with. A good way to create random higher and lower ordered functions is to create a Sum of Gaussians (SoG) function that is initialized randomly with set values of seed, dimension, and number of centers of Gaussians.

Solution:

For this problem, we have two algorithms that at our disposal: Hill-climbing and Simulated Annealing. Hill Climbing is act of climbing a hill, taking step sizes based on the gradient of the function, and effectively stops when the next point to evaluate is less than 0.00000001 from the one that it is on currently. The problem is that this only solves for local hills or **local maximums**.
Therefore, we have Simulated Annealing, which will upon reaching the top of a hill, will try to make bad decisions to move away from the hill and maybe come out onto a global hill or **global maximum**. (or to a smaller hill)

Design:

The Hill Climbing algorithm wasn't difficult to write and consisted of these simple steps:

1. $Initialize\ an\ SoG\ function\ with\ random\ values$
2. $Loop\ do$:
    a. $Print\ current\ location\ (X)\ and\ the\ evaluation\ at\ that\ location\ (G(X))$
    b. $Call\ the\ derivation\ on\ the\ current\ value\ into\ a\ d[x]\ array$
    c. $Loop\ for\ dimensions$:
        i. $Step\ Size\ =\ (.01\ \times\ d[x])$
        ii. $Assign\ a\ neighbor\ array\ the\ step\ size\ +\ the\ current[x]$
    d. $If\ fabs(\ eval(neighbor)\ -\ eval(current)\ )\ <\ 0.00000001$
        i. $Break, we're\ done$
    e. $current\ \leftarrow\ neighbor$

The Simulated Annealing algorithm was more difficult to write, consisting of a **cooling schedule** and had these steps:

1. $Initialize\ an\ SoG\ function\ with\ random\ values$
2. $Initialize\ T\ \leftarrow\ 1$
3. $Loop\ while\ 0\ <\ 100000$:
    a. $Print\ current\ location\ (X)\ and\ the\ evaluation\ at\ that\ location\ (G(X))$
    b. $Loop\ for\ dimensions$:
        i. $Step\ Size\ =\ (.01\ *\ d[x])$
        ii. $Assign\ a\ neighbor\ array\ the\ step\ size\ +\ the\ current[x]$
    c. $if\ G(Y)\ >\ G(X),$
        i. $current\ \leftarrow\ neighbor$
    d. $else$:
        i. $probability\ <=\ e^{(G(Y)-G(X))/T}$
        ii. $if\ probability\ >\ .01$
            1. $current\ \leftarrow\ neighbor$
    e. $T\ *=\ .999\quad \leftarrow\ \boldsymbol{temp\ decrease}$

Data:

I got a ton of data by running scripts to run these algorithms with 100 random seeds, 1-5 dimensions, and 10, 100, 1000, 10000 SoG. I used the first two dimensions to visualize the graphs, and collected data on how many times Simulated Annealing beat Greedy. **FIG A** shows a demonstration of the Greedy step size increase in the y direction. **FIG B** shows the differences in dimensions and Gaussians to show how many times Simulated Annealing beat Greedy, a mean, medium, and maximum. Here we can observe that the more SoG, the less likely Simulated Annealing beat Greedy. The mean shows us that SA beats Greedy marginally, but sometimes a lot as per the max.

FIG A

```
x size:  101
y size:  101
gx size:  333
gy size:  333
sx size:  100000
sy size:  100000
[ 0.413011   0.4178591  0.4229008  0.4281437  0.4335956  0.4392643
  0.4451582  0.4512857  0.4576558  0.4642774  0.4711598  0.4783126
  0.4857458  0.4934693  0.5014935  0.5098291  0.5184869  0.5274779
  0.5368136  0.5465053  0.5565648  0.5670039  0.5778347  0.5890691
  0.6007195  0.6127981  0.625317   0.6382886  0.6517249  0.665638
  0.6800396  0.6949413  0.7103543  0.7262894  0.7427569  0.7597666
  0.7773276  0.7954481  0.8141356  0.8333966  0.8532365  0.8736595
  …
  2.5747967  2.5747967  2.5747968  2.5747968  2.5747969  2.5747969
  2.5747969  2.5747969  2.574797   2.574797   2.574797   2.574797
  2.5747971  2.5747971  2.5747971  2.5747971  2.5747971  2.5747972
  2.5747972  2.5747972  2.5747972  2.5747972  2.5747972  2.5747972
  2.5747972]
```

FIG B

SoG: 10	D=1 sa>g: 0.96 mean: 0.07 max: 3.05 D=2 sa>g: 0.94 mean: 0.33 max: 2.20 D=3 sa>g: 0.97 mean: 0.67 max: 1.80 D=4 sa>g: 0.83 mean: 0.83 max: 1.60 D=5 sa>g: 0.94 mean: 0.93 max: 1.00
SoG: 100	D=1 sa>g: 0.63 mean: 0.31 max: 8.76 D=2 sa>g: 0.93 mean: 0.22 max: 3.05 D=3 sa>g: 0.79 mean: 0.07 max: 2.03 D=4 sa>g: 0.42 mean: 0.21 max: 1.38 D=5 sa>g: 0.57 mean: 0.52 max: 1.45
SoG: 1000	D=1 sa>g: 0.30 mean: 0.63 max: 17.92 D=2 sa>g: 0.67 mean: 0.79 max: 10.31 D=3 sa>g: 0.54 mean: 0.26 max: 4.42 D=4 sa>g: 0.36 mean: 0.22 max: 2.17 D=5 sa>g: 0.35 mean: 0.10 max: 1.64
SoG: 10000	D=1 sa>g: 0.22 mean: 6.42 max: 145.50 D=2 sa>g: 0.20 mean: 2.08 max: 43.63 D=3 sa>g: 0.14 mean: 0.88 max: 13.92 D=4 sa>g: 0.21 mean: 0.64 max: 4.89 D=5 sa>g: 0.24 mean: 0.40 max: 2.18

Visualizing 1-D:

FIG 1

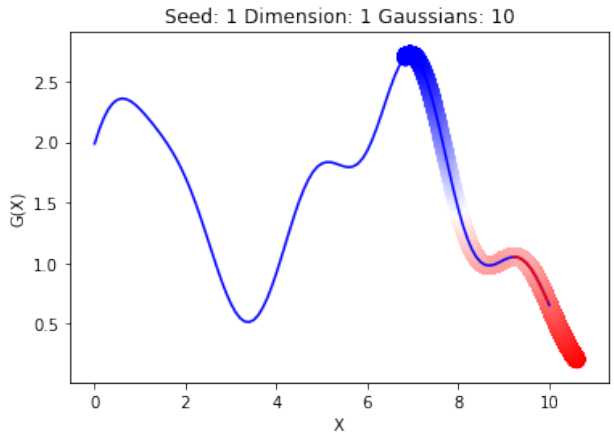


FIG 2

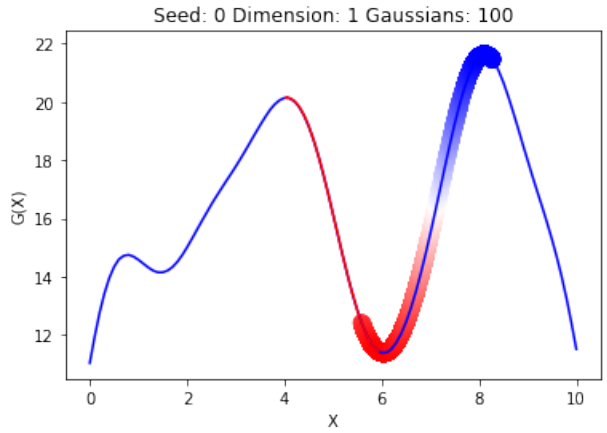


FIG 3

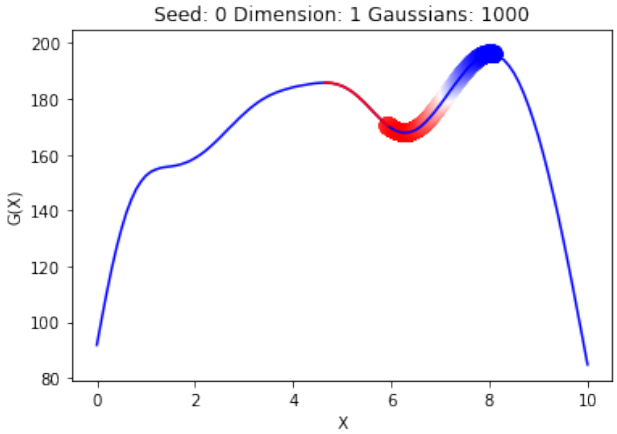
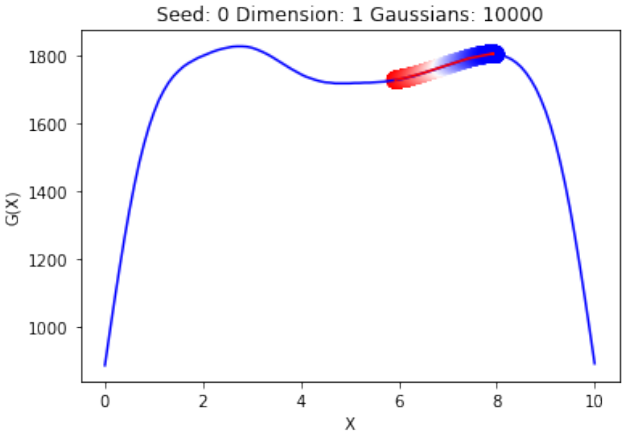


FIG 4



Here we can show some highlights of the Greedy algorithm vs. the Simulated Annealing algorithm. The blue line represents the 1-D function, the red line on top of the blue is the Greedy algorithm, and the red to blue blots are the simulated annealing in which the color of does not necessarily correlate to the temperature

FIG 1: A great case of simulated annealing beating greedy by overcoming a local maximum to find the global maximum.

FIG 2: This time, simulated annealing was smart enough (or lucky) to actually go the opposite way and find the maximum

FIG 3: Same case as FIG 2, but with a higher SoG

FIG 4: This is what all of the higher SoG graphs looked like, a giant plateau with huge valleys in the middle. It would take a lot of luck to get to the maximums there.

Visualizing 2-D:

Now that we've gotten a good idea on 1-D visualizations, 2-D was a bit harder to plot and visualize. **FIG 5** shows the yellow SA beating the red Greedy. **FIG 6** shows SA beating Greedy again, but more than marginally.

FIG 5

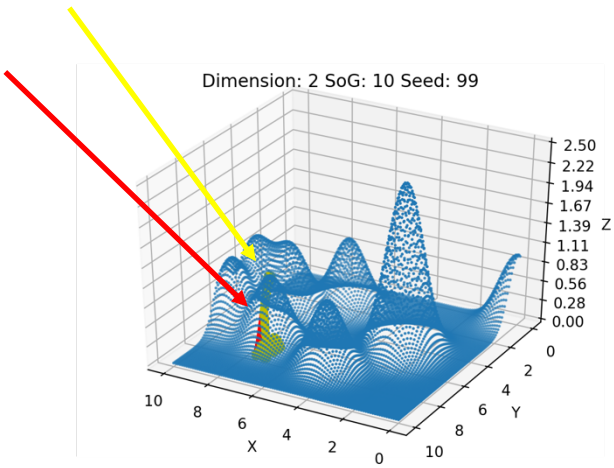
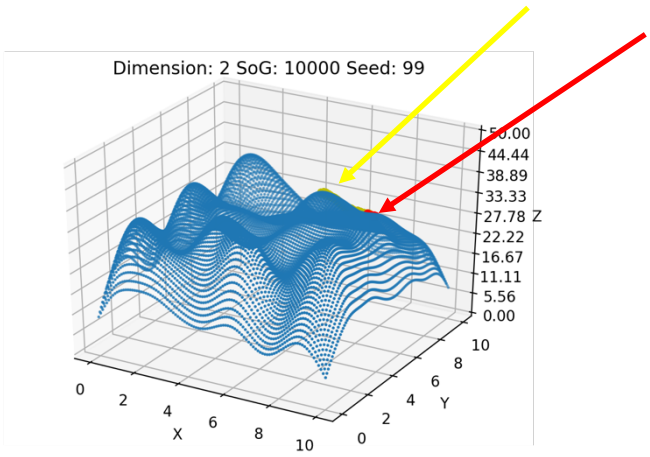


FIG 6



Problems:

Having a SoG of 10,000 on greedy created unexpected loops where an evaluation would jump to extremely high steps and never climb a hill. Ex. If the evaluation was in the 1000's range, 1000 * .01 = 10, plus the step size meaning that we would bounce in-between numbers like 1->9->1->9 etc. This was solved by creating a MAX_ITER of 10,000 and looping against that instead of the .0000001 difference cutoff.

Executing each set with higher Gaussians, especially Simulated Annealing created the problem of evaluations becoming expensive in the time region. On System64, an average execution of a Simulated Annealing at 10,000 SoG would complete in around 2 minutes. 2 minutes * 100 random seeds = some 200 minutes to run. Assigning the evaluations (calling it only once) helped with this.

FIG C shows this. 200 minutes * 5 dimensions = **~17 Hours of computation. (may be due to my implementation.)**

FIG C

```
aam6v@system64:~/School/AI/p2$ time ./final_script.sh
d1:
g 10
g 100
g 1000
g 10000
sa 10
sa 100
sa 1000
sa 10000

real    208m37.923s
user    208m25.016s
sys     0m7.160s
aam6v@system64:~/School/AI/p2$
[0] 0:bash*                                          "system64.cs.mtsu.edu" 15:00 10-Oct-17
```

Optimizations:

My Cooling schedule for Simulated Annealing was tuned by running scripts to change the Limit accepting of the heat. I used an initial heat of **1** and multiplied by **.999** each loop to get a linear decrease of heat for the probability. My limit ended up being extremely low - **.01**. After a quick plot in Excel I came up with a linear equation: $T = -1E-05I + 1$ where **T** is the temperature and **I** is the iteration.