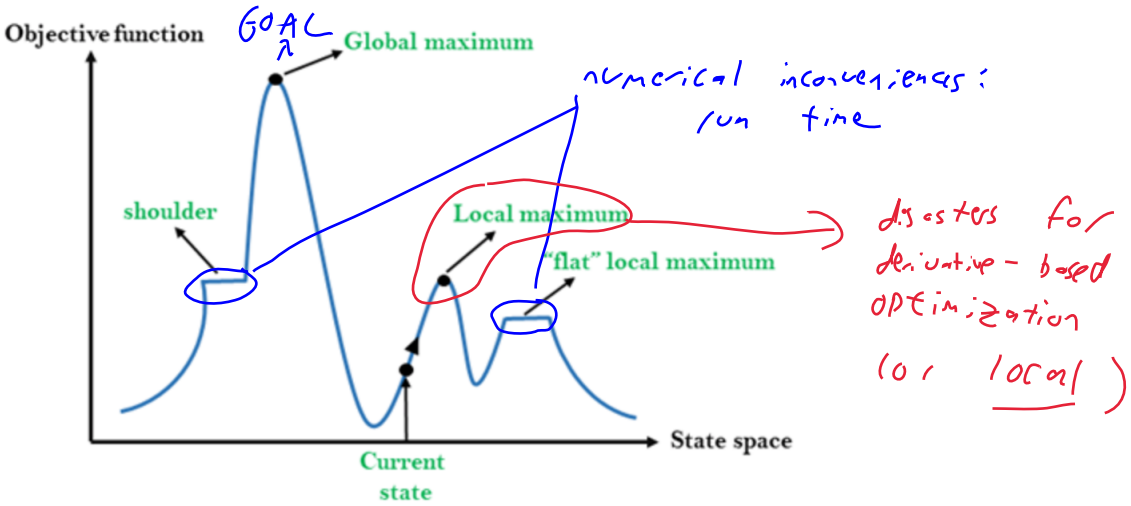
Sept 21: Local Search



Announcements and To-Dos

Announcements:

1. HW 2 due ~~Friday~~ Monday!: lots of searching!

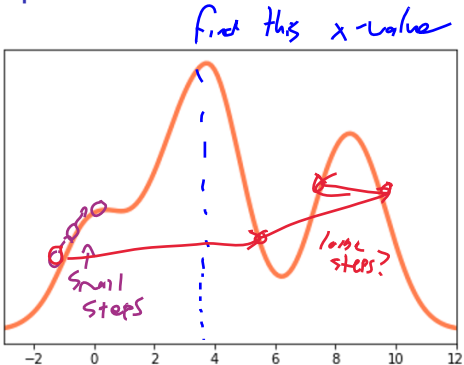
Last time we learned:

1. Some theory on heuristics.

$A*$ Recap: A: one step at a time

1. We concluded that A^* is **optimally efficient** for any given heuristic - and we wanted them to be consistent, although admissible was an easier criteria to "eyeball."
2. We theorized 3 main sources for heuristics:
   - 2.1 from relaxed problems, with more movements available
   - 2.2 from sub-problems, with simpler goal states
   - 2.3 from experience: combining properties of tasks whose solution path length is well-known and making a model
3. The first two of these were inherently admissible, the third slightly worrying, but sounded powerful.
4. All heuristics followed a general idea: the closer the heuristic is to the real value (or the larger the heuristic, if admissible), the better. However, we don't want to lose any time savings that taking less steps/opening less nodes provides if our heuristic takes too long to compute!

Optimization



Lay me open nodes in our space: irrelevant

find: best soln / best action matter

Suppose our path to the goal doesn't matter to us: only that we eventually get there.

In many problems, we're tasked then with finding the "best" mathematical object, which typically means something like "maximum probability" or "minimum error/cost".

Many Variables

From an agent standpoint, this isn't a function with only one-dimensional input, though: we're possible interested in inputting entire strings of possible agent actions and choosing the best functional output between them. = path through sample space

More broadly: optimizing a function in the presences of lots of data and even more so, **lots of features**, leads to challenging direct computation.

- The direct solutions rely on multivariate calculus, and lead slow, memory-intensive, and occasionally numerically unstable linear algebra
- More commonly, we use an iterative method that *converges* after many steps to the (a?) desired value.

General Iterative Algorithms

In general, iteration schemes are the algorithmic version of "guess-and-check."

1. Start with a guess
2. Ask how bad our guess was, and update it in a way that's informed by the problem (and has a high probability of being a better guess)
3. Repeat until your guess has converged to something close to the correct answer

For an agent, this may be equivalent to following paths on the state space and choosing actions probabilistically.

1D Algorithm

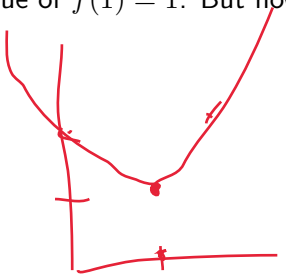
Suppose we wanted to find the global minimum of the function $f(z) = z^2 - 2z + 2$.

1D Algorithm

upward parabola

Suppose we wanted to find the global minimum of the function $f(z) = z^2 - 2z + 2$.

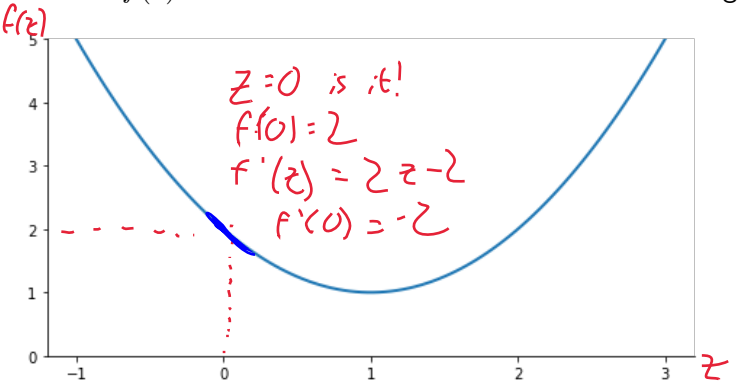
Rewritten as $f(z) = (z-1)^2 + 1$, we should notice that the minimum is at $z = 1$ and is the value of $f(1) = 1$. But how would we find that with an algorithm?



1D Algorithm

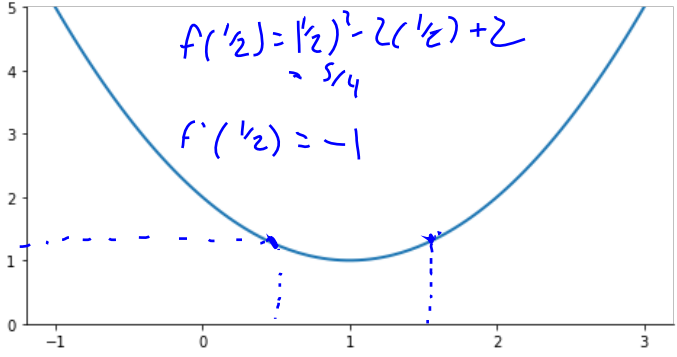
Suppose we wanted to find the global minimum of the function $f(z) = z^2 - 2z + 2$.

Rewritten as $f(z) = (z-1)^2 + 1$, we should notice that the minimum is at $z = 1$ and is the value of $f(1) = 1$. But how would we find that with an algorithm?



1D Algorithm

Suppose we wanted to find the global minimum of the function $f(z) = z^2 - 2z + 2$.



$f(1/2) = (1/2)^2 - 2(1/2) + 2$

$= 5/4$

$f'(1/2) = -1$

$f'(z) = 2z - 2$

guess: $1/2 + 1 = 3/2$

$f(3/2) = 5/4$

$f'(3/2) = 1$

First guess: $z^{(0)} = .5$. What do we do next? Move that direction? How far? Maybe one unit?

Gradient Ascent/Descent

The idea of following the derivative to find the maximum or minimum value of a function is called gradient ascent or gradient descent. It requires:

1. The ability to calculate the slope of the function we're trying to min/max
2. An idea of how large of steps to take; a step size or *learning rate*

$$\underbrace{z^{(k+1)}}_{\text{new guess}} = \underbrace{z^{(k)}}_{\text{old guess}} - \left(\underbrace{\nu}_{\text{step size}} \underbrace{f'(z^{(k)})}_{\text{step direction}} \right)$$

ν (constant?) or $\nu(t)$ = # of steps $\nu(n)$

$|f'(z^{(k)})|$

Gradients in Regression

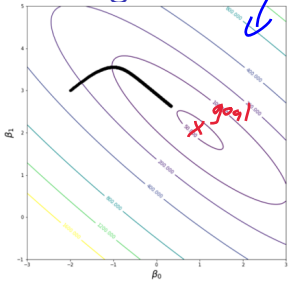
The optimization is the same in higher dimensions: we move "downhill" to the region of lowest error or cost. Now, though, downhill is a partial derivative: it says we need to move in the downhill direction for all of our relevant inputs:

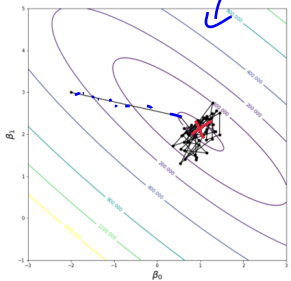
The size of the movement could be proportionate to the slope, or generally just small enough to be safe so we don't jump "past" the region.

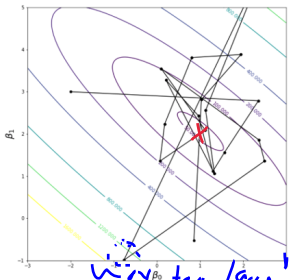
Learning Rates

too small

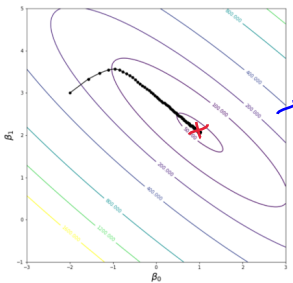
too large







way too large



just right

Agent Steepest Ascent

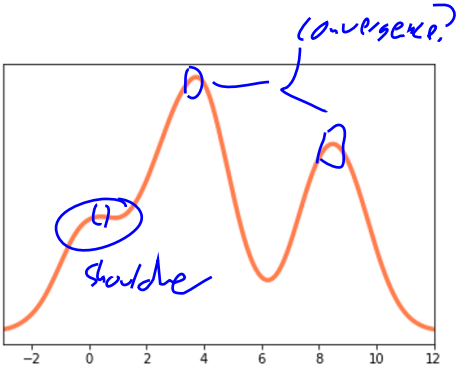
Consider applying this approach to an agent. The algorithm might look like:

1. Use only the current node/state
2. Consider only moves to neighbors
3. This would be *very* memory efficient!
4. Moves to the best state per some "objective" function. Sounds a lot like the greedy best-first heuristic algorithm!
5. We call this a "steepest ascent" approach to optimizing over some objective function.

(heuristic when we did best-first)

Hill Climbing

How do the rest of the movements translate to an agent? And how do we ensure that we got a *global* maximum, not just an local one?



1. What do we do if we hit a shoulder or local max?
2. Can we move sideways? Can we skip nodes along a path?
3. What is the "convergence" criteria on a graph?
4. This tends to lead to *stochastic* hill climbing.
5. We may also include random *restarts* to ensure that we get the same result from different initial conditions, or repeatedly from the same condition.

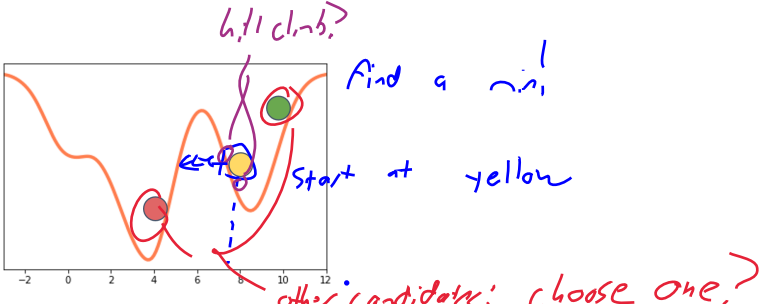
Hill Climbing

```
class state:

    # we need to be able to keep track of states and their associated
    # objective function values


class problem:

    # We want to include everything that defines the problem here:
    # - initial state
    # - current state
    # - what is the objective function we're trying to maximize/minimize?
    # - what are the choices of action that we have?
    # - what do we need to know to select our action?


def hill_climb(problem, number of iterations):

    for t in some number of iterations:
        1. get a list of our available moves
        2. which move optimizes the objective function?
        3. do that move; update our state
        4. possible goal/convergence check
```

An Application: Annealing

Visualization of what happens when you heat treat (e.g. Anneal) a metal Video: `https://www.youtube.com/watch?v=xuL2yT-B2TM`

An Application: Annealing



When you heat and cool a metal, the size of the crystals formed iteratively reshape and may have their defects reduces. This is the idea of the algorithm we will explore: repeatedly check a new state (e.g. a heat-cool cycle) for an improvement or not, and move there most of the time if so.

Simulated Annealing

Simulated Annealing implements this process thusly:

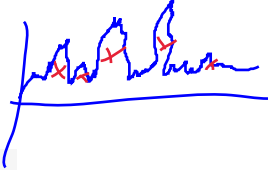
1. Propose random moves to *nearby* neighbor (*not* necessarily adjacent)
2. Accept that move with some probability that scales with:
   - 2.1 "temperature," T or some kind of estimated cost to make that movement. We tend to let the movements be very large initially, then as we take more and more steps make the movement more and more *local* (or $T \to 0$)
   - 2.2 How "good" the move is: accept without question if the move is clearly much better than our current state, accept with some probability < 1 if it's worse. May scale with T as well

What does this look like? See here:

https://en.wikipedia.org/wiki/Hill_climbing#/media/File:Hill_Climbing_with_Simulated_Annealing.gif

Simulated Annealing

find global min/max:



```
def schedule(time):
    '''some sort of mapping from time to temperature, to represent how we should be
    "cooling off" - that is, accepting wacky solutions with lower and lower probability'''
    # math goes here
    return temperature

def simulated_annealing(problem, some number of iterations):

    for t in some number of iterations:
        #1. update the "temperature", T(t) = schedule(time)
        #2. which moves can we make from the current node?
        #3. pick a random move
        #4. calculate difference in objective function between
        #   proposed new state and the current state (deltaE)
        #5. if proposed new state is better than the current state,
        #   then accept the proposed move with probability 1
        #6. otherwise...
        #        ACCEPT the move with probability exp(-deltaE/T(t)),
        #     or REJECT with prob 1-exp(-deltaE/T(t))
```

(& propose)

Simulated Annealing

there are many options for the $T(t)$ & Prob $(n \to n')$ functions Temp(time)

We will do this on an upcoming notebook. For that, we will somewhat arbitrarily choose, for constants C and p:

$$T(t) = \frac{C}{(t+1)^p}$$

Note that this is a function that will decay to 0 as time t increases, which is important for these problems.

$\max(p_{accept}, 1)$

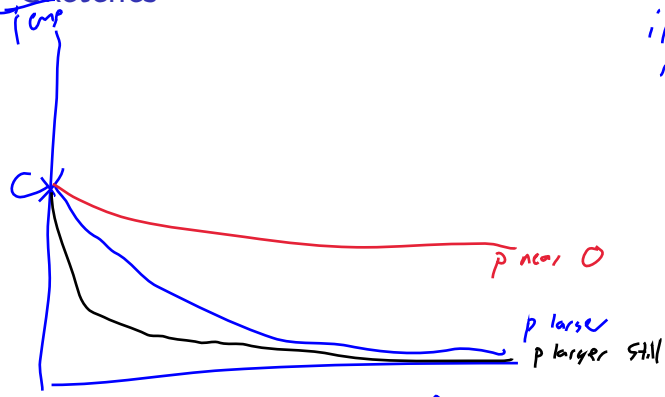
$$p_{accept} = \exp\left(\frac{\Delta E}{T(t)}\right) = e^{\left(\frac{\Delta E}{T}\right)}$$

$\Delta E > 0$: new state > old state

What happens here as $T \to 0$:

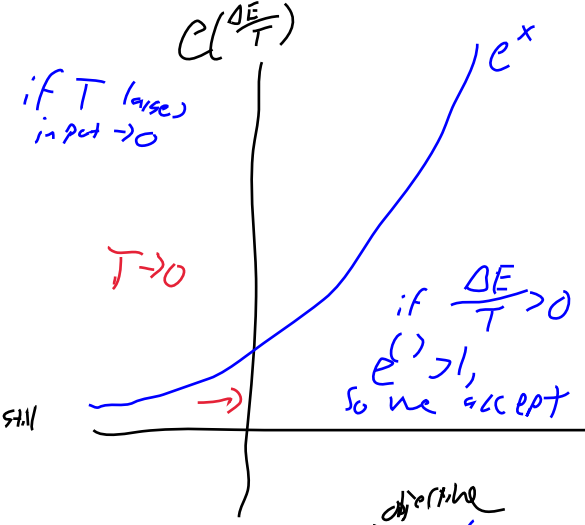
1. ...if Δ E is positive, or the new state is better than our prior state?
2. ...if Δ E is negative, or the new state is worse than our prior state?

Sketches



$(t+1)^p \to (\#)^p \quad p > 0$

$$T(t) = \frac{C}{(t+1)^p} \qquad p > 0,$$



$$p_{accept} = \exp\left(\frac{\Delta E}{T(t)}\right)$$

Moving Forward

- This Week:
  1. nb day Friday
- Next time: Local Search!