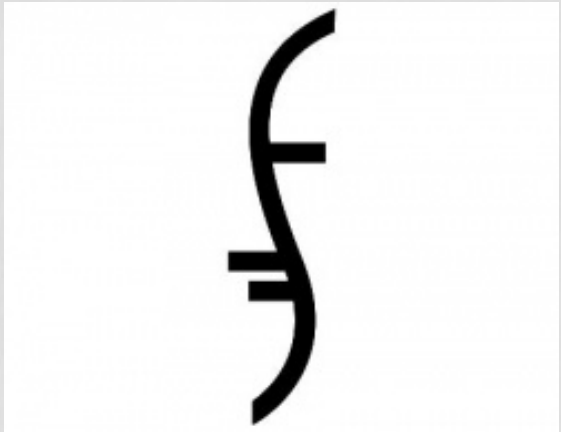


GENETIC ALGORITHMS

OVERVIEW

- Overview
  - Good when:
    - Hard problem, too many possibilities to "brute-force" guess.
    - Easy to evaluate solutions
  - Large number of initial random "guesses"
  - Use Darwinian Evolution to find new solutions.
    - Carry-over's, mutations, mating, [immigrants?]
- Genetic **Algorithm**:
  - Finds the best guess based on a “fitness” score
  - Similar in goal to Simulated Annealing
- Genetic **Programming**:
  - Generate a program to solve the problem
  - Expressed as a tree (side-note Lisp and language grammars)

ENCODING THE PROBLEM

- Always the hard part.
- Classically a bit string
- E.g. http://alglobus.net/NASAwork/papers/Space2006Antenna.pdf
  - Jason's guess: a string of
    - 2-bit opcodes
      - 00 = fwd
      - 01 = rotateL
      - 10 = rotateR
      - 11 = roll
    - 6-bit amounts
      - mm for forward
      - angles for the other 3.
- Evaluate fitness
  - Also hard!
  - Need a number score based on a guess
  - Jason's guess on NASA's approach:
    - some kind of super-awesome flux-simulator
    - more points for a better antenna reception

NAÏVE: GRADIENT DESCENT

- A simpler method than SA or GA
- Starts with an initial guess to the problem, s_0
- Calculates the gradient at s_0 and moves in the direction of increasing gradient by some Δ to get s_1.
- …and so on…
- [Show a 1d example]
- Problems:
  - Local maxima
  - Hard to determine size of Δ

BETTER: SIMULATED ANNEALING

- "Randomized gradient descent"
- https://en.wikipedia.org/wiki/Simulated_annealing
- Pseudo-code:

```
s⃗ = random_guess()
i = n
while i > 0:
        T = temperature(i, n)     # Decreasing
        s_alt⃗ =get_neighbor(s⃗, T)
        if probability(s⃗, s_alt⃗, T) < random.uniform(0, 1):
                s⃗ = s_alt⃗
        i--
```

- The probability function is based on:
  - How far apart are the neighbors? Farther away = less likely
  - The temperature. Lower temperature means far away neighbors are less likely.
  - How high is the current value? Higher values mean we're more likely to stay put.

BETTER: GENETIC ALGORITHM

- “Survival of the fittest”
- Pseudo-code

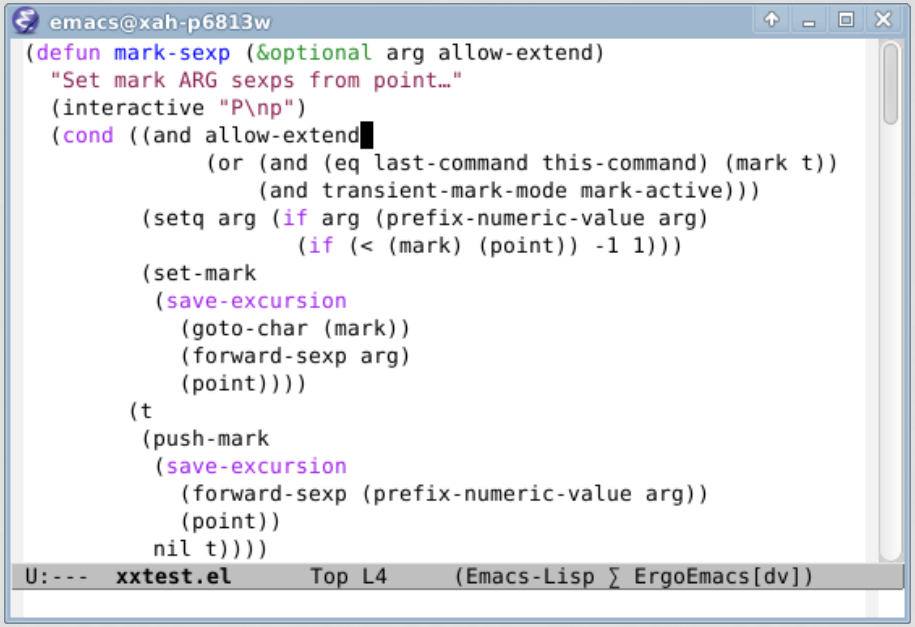
```
gen = []
for i in range(gen_size):
    gen.append(random_guess())
while we haven’t found a good enough solution:
        new_gen = []
        # Normally mate_percent + carryover_percent = 1.0
        for i in range(len(gen) * mate_percent):
            new_gen.append(mate(pick(gen), pick(gen)))
        for i in range(len(gen) * carryover_percent):
            new_gen.append(pick(gen))
        for i in range(len(new_gen)):
            mutate(new_gen[i])
        gen = new_gen
```

MORE DETAILS

- pick needs to be a “biased” picker
  - More fit = more likely to be picked
  - Do we allow the same individual to be picked multiple times?
  - Ideas:
    - Sort and then a gaussian random
    - Tournament selection
- We could also allow some “immigrants” – new random guesses
- Mutation is an attempt to avoid local maxima
  - Should have some (smallish) chance of modifying an entity.
- Mating should combine the genes of both parents

AWESOME: GENETIC PROGRAMS

- Instead of solving the parameters for a problem, we evolve a program that solves the problem
- Programs as trees
  - Lisp
  - Language grammars: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html



OPERATIONS IN G-PROGRAMMING

- Fitness is often calculated by "emulating" the program
  - Often (especially initially) you may get weird programs that don't do anything.
  - It is important that your fitness function give some "pity-points" for doing something
- Mutation
  - Replace one of more sub-trees in a tree with a randomly generated tree
- Mating
  - Select a splice point in parent A and remove it, replacing it with a randomly selected sub-tree from parent B.

DON'T BE A MAD SCIENTIST!

- We often have hunches on how best to solve the given problem
- Avoid the temptation to make those entities that match our expectations more fit
- Genetics work best when you let it do its thing…naturally.