

Sept 23: Genetic Algorithm

Let's recap last time! In simulated annealing, what were the purposes of:

1. The “temperature” function?
2. The “Probability of accept” function?

Announcements and To-Dos

Announcements:

1. HW 2 due Monday: lots of searching!

Last time we learned:

1. About a couple of local search (min/max) techniques.

Minute form commentary:

1. On doing more code
2. On more applications (when do you use alg x , how is this even "AI", etc.)

Brief HW2 commentary

You can create your own implementation of the searching algorithms if you wish, but you are still expected to get the desired outputs in an appropriate format: lists of the nodes chosen for the final path, and the cost of the path.

If your dictionary is given to you in an order (that might *not* be alphabetical), that's fine! You can just use that order for your BFS/DFS expansions.

Your textbook has pseudo code implementations of each and every one of these algorithms! Use them to start you off!

Piazza pseudo code = ☺

HW Helpers

The path function is expecting you to maintain some sort of self-referencing dictionary. For example: $\{A: \text{NONE}, B:A, C:B\}$ would be a dictionary whose $\text{PATH}(C)$ would return B, A.

$\{A:A\}$
 $\{B:A\}$
 $\{C:B\}$

$C \leftarrow B \leftarrow A$

You can use that type of dictionary for more things than just a list of prior states: feel free to include the prior state information in either your explored set, frontier set, or both!

Imagine a short pseudocode for BFS:

$\{ \}$

$\{A: \text{None}\}$

$\{A: \text{None}\}$

$\{B:A, C:A\}$

1. Maintain two dictionaries: explored and frontier.
2. Loop as long as there are things in the frontier: pick the oldest/first element in the frontier. Check its neighbors from the input state graph. Then, for each neighbor

→ DFS: newest!

of what we explore...

2.1 Check if it's the solution. If it is, get its path.

if DFS: remove prior version

2.2 Add it to the frontier (if it's not already there) Include the parent (for path to work)

Annealing and Hill Climbing Recap

We're considering trying to create optimizations of agent behavior where the inputs are often very complicated and diverse: e.g. problems with extremely high or even *continuous* “branching factors” of a sort.

1. We use an iteration scheme:

- 1.1 Start with a guess

→ score/objective/evaluation

- 1.2 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)

- 1.3 Repeat until your guess has converged to something close to the correct answer

(or goal,?)

Hill Climbing Recap

Scheme #1 is Hill climbing: this behaves much like a greedy best-first algorithm.

1. Use only the current node/state
2. Consider only moves to neighbors (Small neighborhood)
3. As with the full-greed approach, this maintains no active memory of past states, and would be very memory efficient!
4. This can be appealing if any given state has significant memory burdens associated with it.
5. For multivariate functions with continuous inputs, this is the same concept as taking a derivative and walking in the direction of the (steepest) slope.

Annealing Recap

Scheme #2 is Simulated Annealing: *maybe not so nearby...*

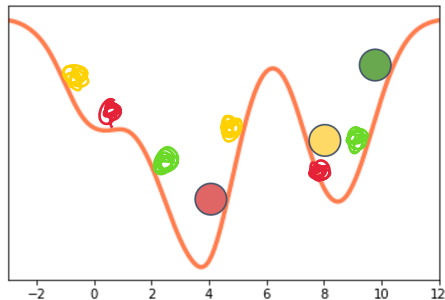
1. Propose random nodes to *nearby* neighbor (*not* necessarily adjacent).
2. Accept that move with some probability that scales with:
 - 2.1 “temperature,” or some kind of estimated cost to make that movement. We tend to let the movements be very large initially (*jumps*), then as we take more and more steps make the movement more and more *local* (or $T \rightarrow 0$)
 - 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
*(by our heuristic/
objective function)*

Annealing Recap

Scheme #2 is Simulated Annealing:

1. Propose random nodes to *nearby* neighbor (*not* necessarily adjacent). This is one of the main purposes of the *Temperature* function: it is a function that decays over time. Think of it a function whose output units are *distance between states*. As we force $T \rightarrow 0$ as we iterate, we propose new states that are **closer** to where we currently are.
2. Accept that move with some probability that scales with:
 - 2.1 For each new state proposal, we compare their functional **output**, sometimes called *energy*. It's our estimate/calculation/heuristic/etc. for how "good" each state is.
 - 2.2 *Then*, we move to that new state if it's better than our current one. Or we move there if it's worse, but not *that much* worse. This requires a new function that we're calling the p_{accept} function.
 - 2.3 Over time, we dial $p_{accept} \rightarrow 0$ for states that were worse. This makes sure that we're only **climbing** instead of **jumping** for the last steps of our iteration.

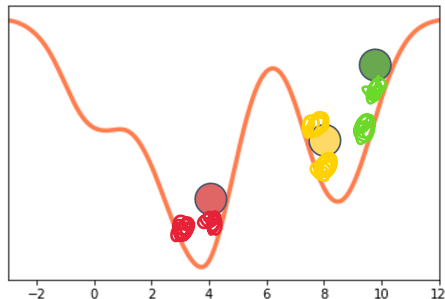
Annealing Recap



1. High “Temperature” tells us to try out states all over the function and explore them.

We will do this Friday in lecture as a notebook: it’s posted to the course schedule already, so check it out!

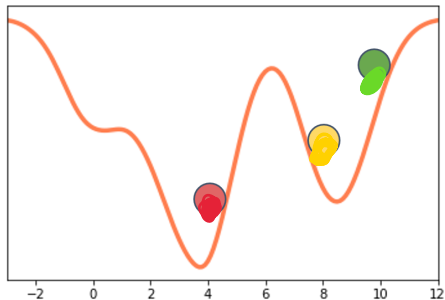
Annealing Recap



1. High “Temperature” tells us to try out states all over the function and explore them.
2. At middle “Temperature,” we bounce around some, but also try to slowly work our way *downwards* (or upwards, depending on the problem)

We will do this Friday in lecture as a notebook: it’s posted to the course schedule already, so check it out!

Annealing Recap

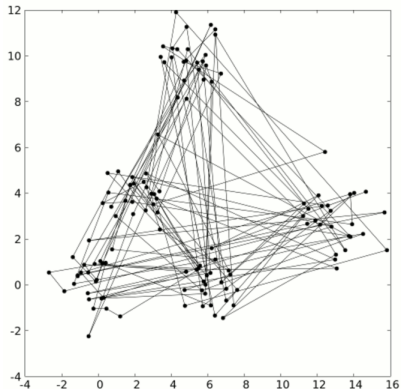


1. High “Temperature” tells us to try out states all over the function and explore them.
2. At middle “Temperature,” we bounce around some, but also try to slowly work our way *downwards* (or upwards, depending on the problem)
3. At low temperatures, only hill climb!

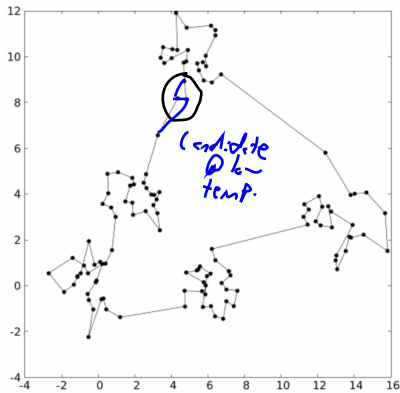
We will do this Friday in lecture as a notebook: it’s posted to the course schedule already, so check it out!

Local Search Application

E= 852 T=125



E= 94 T= 1



input:
Path/
Sequence of
all n nodes

Sol: $[1, 2, 5, 7, \dots]$
 $1 \rightarrow 2 \rightarrow 5 \rightarrow \dots$

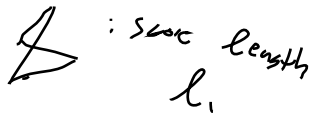
The “traveling salesman” problem is an optimization problem. Find the shortest possible route that passes through every node exactly once, ending where we started.

Traveling Salesman

How might this work?



Initial: $[0, 1, 2, 3]$



Start with a random list (a permutation) of the nodes. This implies a full path, with edge lengths we could calculate. Then loop:

1. Use the "temperature" function to propose a new *path*. Do this by creating a new path generated by making a few "swaps" of elements in the old path.
 - 1.1 When temperature is high, do a good number of swaps of elements that might be quite far away in the list.
 - 1.2 When temperature is medium, do only a few swaps of elements, and make it less likely to pick two elements that are far apart.
 - 1.3 When temperature is low, only do 1-2 swaps at a time, and only propose elements that are nearly adjacent in the list.
2. Any time we make a swap, we can compute the *energy* of the new path: for the salesman, it's just the total length of the path.

Random Restarts

Both a simple hill climb and annealing have some risk of settling in at the *wrong* maximum: a local max rather than a global max. The solution is in repetition:

1. Conduct a series of hill-climbing searches with a randomized starting point.
2. If your random restart initial points end up all being close together, this method may not be any more effective than basic hill-climbing, but it makes us more confident in our results!
with our initial random guesses
3. You want to cover the space as much as you can.
4. Cost is a constant multiple of hill-climbing - not much more expensive.

Local Beam Search



A *local beam search* tries to combine elements of random restarts - multiple starting seeds - with those of annealing.

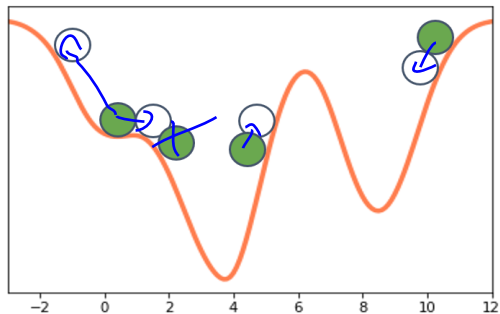
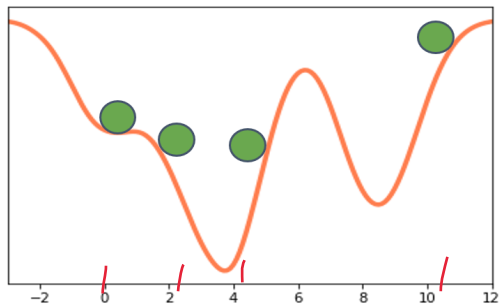
Start with k initial states to explore. Then iterate:

1. Propose **all** ^(or neighbors) successors to each of the k states. (If movement is continuous, have some step size/jitter to make this finite!)
2. Pick k of those random successors to keep exploring in the next step
 - 2.1 Best k ?
 - 2.2 Random k ? Random k that lean towards better ones?

We may call these iterates **generations** of states... and we expect the **generations** to gravitate towards local and global optima!

Beam Generations

$k=4$



Genetic Algorithms

“The fact that life evolved out of nearly nothing, some 10 billion years after the universe evolved out of literally nothing, is a fact so staggering that I would be mad to attempt words to do it justice.” - Richard Dawkins

Darwinian Natural Selection posits 3 key principles that must be in place for evolution to happen:

1. Heredity - process by which children receive the properties of their parents
2. Variation - Variety of traits present in population or a means with which to introduce variation
3. Selection - Some members have opportunity to pass on genetic info and some don't; survival of the fittest

Genetic Algorithm Outline

Today we consider a variant of local beam search that carefully controls how to generate new successors. Specifically, we generate them by **breeding** two predecessor (**parent**) states.

SETUP

Step 1: **Initialize:** Create a population of N elements, each with randomly generated DNA.

features/inputs

DRAW

Step 2: **Selection:** Evaluate the fitness of each element of the population and build a mating pool.

Step 3: **Reproduction:** Repeat N times:

- Pick two parents with probability according to relative fitness.
- Crossover — create a “child” by combining the DNA of these two parents.
- Mutation — mutate the child's DNA based on a given probability.
- Add the new child to a new population.

Step 4. Replace the old population with the new population and return to Step 2.

Genetic Algorithms

Initialization:

1. Begin with a **population** of k randomly generated states.
2. Each individual must somehow be represented as a **string** (like DNA) from some finite alphabet.

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198					
87932185					
12197835					
90271049					

Genetic Algorithms

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%			
87932185	15	27%			
12197835	25	45%			
90271049	6	10%			

in 100
 output
 fitness
 [0, 1].
 ϵ to 1.

Genetic Algorithms

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times

Parents

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
#3 28439198	10	18%	#1 12197835		
#2 87932185	15	27%	#2 87932185		
#1 12197835	25	45%	#1 12197835		
90271049	6	10%	#3 28439198		

Genetic Algorithms

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times
3. "Breed" those pairs, in some random way picking traits/values from each.

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%	12197835	12197185	
87932185	15	27%	87932185	87932835	
12197835	25	45%	12197835	28437835	
90271049	6	10%	28439198	12199198	

Genetic Algorithms

Then we loop over generations:

1. Each specimen can be evaluated: it has a fitness/survival level, and from that an estimated probability that it's "good enough" to reproduce.
2. From the population eligible for reproduction, select pairs of parents, which may appear multiple times
3. "Breed" those pairs, in some random way picking traits/values from each.
4. Add some added randomness to open truly new states by **mutating** strings with some small, independent probability

Specimen	Fitness	P(Reproduce)	Breeding Pop	Crossover	Mutation
28439198	10	18%	12197835	12197185	13197185
87932185	15	27%	87932185	87932835	87932035
12197835	25	45%	12197835	28437835	28437835
90271049	6	10%	28439198	12199198	92199198





Genetic Algorithm Addendum

One concern with the genetic algorithm: what if the crossover and mutated children don't even satisfy the problem to begin with?

Consider the “N-queens” problem: how many “Queen” pieces in chess can we add to a chess board? Queens can not share a column, row, or diagonal (45 degrees) with another queen.

To the right is a state that constitutes one of the solutions to the 4-Queens problem, that we might represent as

STATE = [2,0,3,1].





	x_0	x_1	x_2	x_3
0				
1				
2				
3				

We could consider using the genetic algorithm: take the string 2031 and breed it to look for *more* solutions. Or take a string like 1234 and use it to look for a solution at all.

Genetic Algorithm Addendum

Suppose our algorithm generated a mutated or child state of $\text{STATE} = [1,0,3,1]$. Since any string with a repeating value represents a shared row, we should immediately discard this state, or repair the “broken chromosome” at the end of it.

If we modify $\text{STATE} = [1,0,3,1]$ to $\text{STATE} = [1,0,3,2]$, this is a technique known as constraint satisfaction.

	x_0	x_1	x_2	x_3
0				
1				
2				
3				

We may often embed these types of corrections into the objective function. Our objective function might be trying to minimize the overlap between queens, but we could also force it to only consider strings that are permutations of $[0,1,2,3]$ with no redundant rows.

We could also just modify the reproduction function itself to somehow only allow “feasible” states: crossovers would involve swapping locations of characters rather than e.g. copying substrings.

Moving Forward

- ▶ This Week:
 - 1. nb day Friday
- ▶ Next week: Games!