Andrew Taylor en605.645 9/12/24

Module 3: Local Search Self-Check

General.

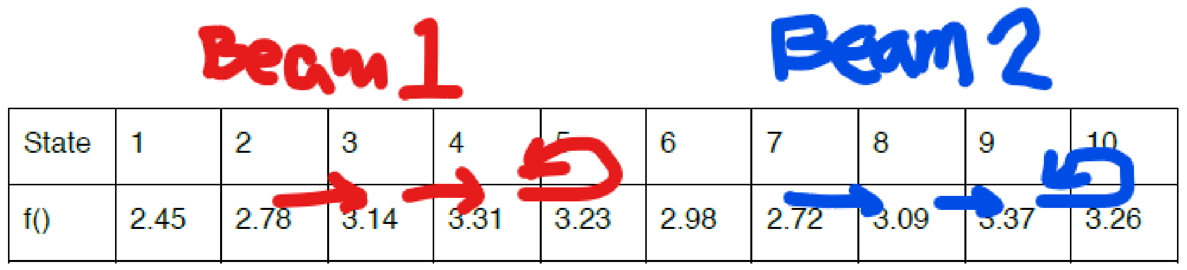
1. If the state is 5, in the given example the successors(5) returns states 4 and 6.
2. If the state is 3, find-best-child(3) will return the child with the highest value which in this case is state 4 where f() = 3.14.

Hill Climbing.

1) If we start randomly with state 7 hill-climbing, line 2 will store 2.72 and line 4 will find state 8 at 3.09 as the best child. Continuing the loop, line 7 takes us to 8 current, then line 4 finds state 9 at 3.37. Line 7 takes us to state 9. The top of the loop at line 4 finds state 10 as the best child at 3.26, but proceeding to line 5 that candidate_value is lower than the current value so it returns state 9's value of 3.37. In this case it happens to be the global maximum.
2) Starting with state 2, the algorithm proceeds right to the best child, again right to the best child, then returns the current state there because the best child is lower than current. Here we are stuck at a local optima.
3) There are many ways to improve hill-climbing to avoid local optima, including simulated annealing, random restart, stochastic hill-climbing, etc. Stochastic hill-climbing is interesting because successors are selected at random but biased towards better neighbors. There's also beam search that expands multiple candidates at once, selecting the top candidates (beams) thus considering multiple solutions simultaneously. The improvements are not guaranteed. Random restart can escape local optima, but can also get stuck. Beam search improves chances but all the beams could get stuck at local optima too.

Beam Search.

1) If we start with k=2 and the states are 2 and 7 the beams proceed thusly. Please forgive me it's easier for me to just draw this out to show:

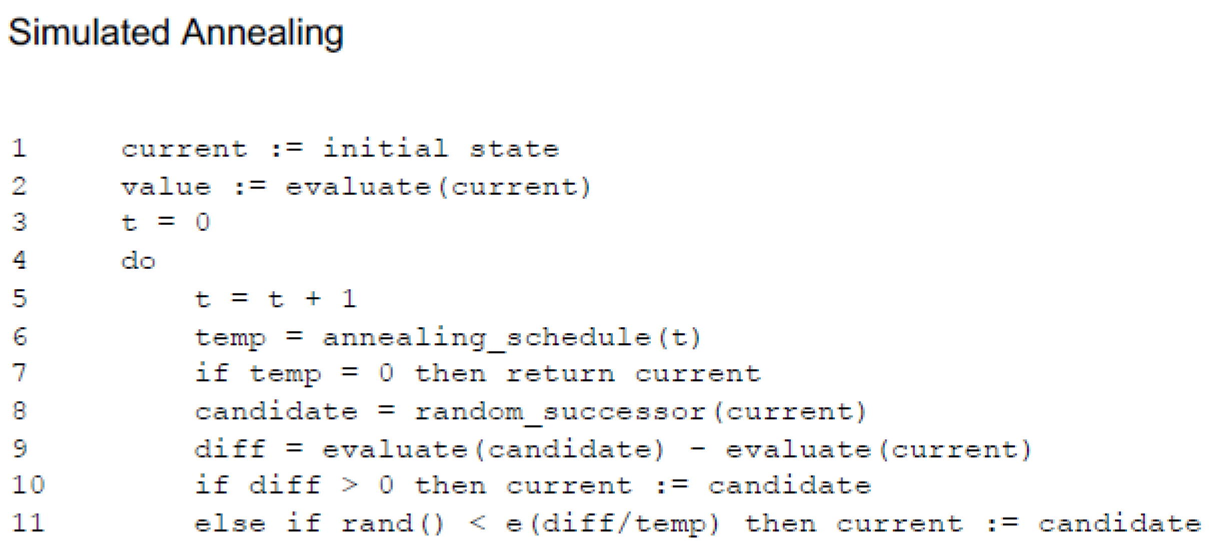


State	1	2	3	4	5	6	7	8	9	10
f()	2.45	2.78	3.14	3.31	3.23	2.98	2.72	3.09	3.37	3.26

Using beam search, if we have these random starts (2 and 7) we find the global maxima with 7. It works.

Simulated Annealing.

1) If we are in state 4, simulated annealing might permit further local search in this way (let's look at the pseudocode):



Simulated Annealing

```
1    current := initial state
2    value := evaluate(current)
3    t = 0
4    do
5        t = t + 1
6        temp = annealing_schedule(t)
7        if temp = 0 then return current
8        candidate = random_successor(current)
9        diff = evaluate(candidate) - evaluate(current)
10       if diff > 0 then current := candidate
11       else if rand() < e(diff/temp) then current := candidate
```

Simulated annealing escapes local optima (line 11) by accepting inferior moves as long as a parameter called "temperature" remains above zero, with a certain probability function governing that choice. The temperature here is "temp" and it is started with an annealing schedule which is usually something like

$T(t) = \frac{T_0}{1 + \beta \cdot t}$ where T_0 starts high, beta is some optional scaling factor, and t is the time step. This creates an inverse decay. This is the annealing schedule covered in lecture. (line 6)

If we are in state 4, then the algorithm could escape the local optima at 4 if the temperature has not yet reached 0. Basically the algorithm bounces around accepting moves that climb a hill (lines 9 and 10) or, if that is not available, a random successor (assigned line 8, accepted line 11) which may be a neighbor that leads to an escape.

Genetic Algorithm.

1) Since the classic genetic algorithm uses an encoding into a chromosome of bits, we can represent a problem with three variables that can take values 0 to 7 in a byte. Some individual with the value of 678 would look like [[0, 1, 1, 0], [1, 1, 1 , 0], [0, 0, 0, 1]] when sotred as a list of lists.

2) The phenotype is the form of the variable that we can actually use in the end after search, this is “678”.
3) The genotype for an individual that is [4,2,5] would be [[0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 1, 0]].
4) In the pseudocode you do not need to go from phenotype to genotype because the code starts with a random population. It is generated after the encoding is decided.
5) A population of three individuals could look like:
    a. [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]] = “000”
    b. [[0, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 0]] = ”777”
    c. [[0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]] = “222”

Crossover and Mutation

a. The children if the gene index is 5 are:
    i. 0101010111110110
    ii. 101011011001000
b. The children if the index is 12 are:
    i. 101001011111000
    ii. 010111011000110
c. If the gene index is zero there is no crossover the son and daughter switch places.

    a. A mutation at location 9 with symbol zero results in 01201120020221110
    b. A mutation at location 2 with symbol 2 results in: 02201120120221110
d. Mutation does happen at 0.0237 because it is below the mutation rate of 0.05.
e. The crossover does not occur in the first case and does in the second because rand() must generate a number below the rate to create crossover.