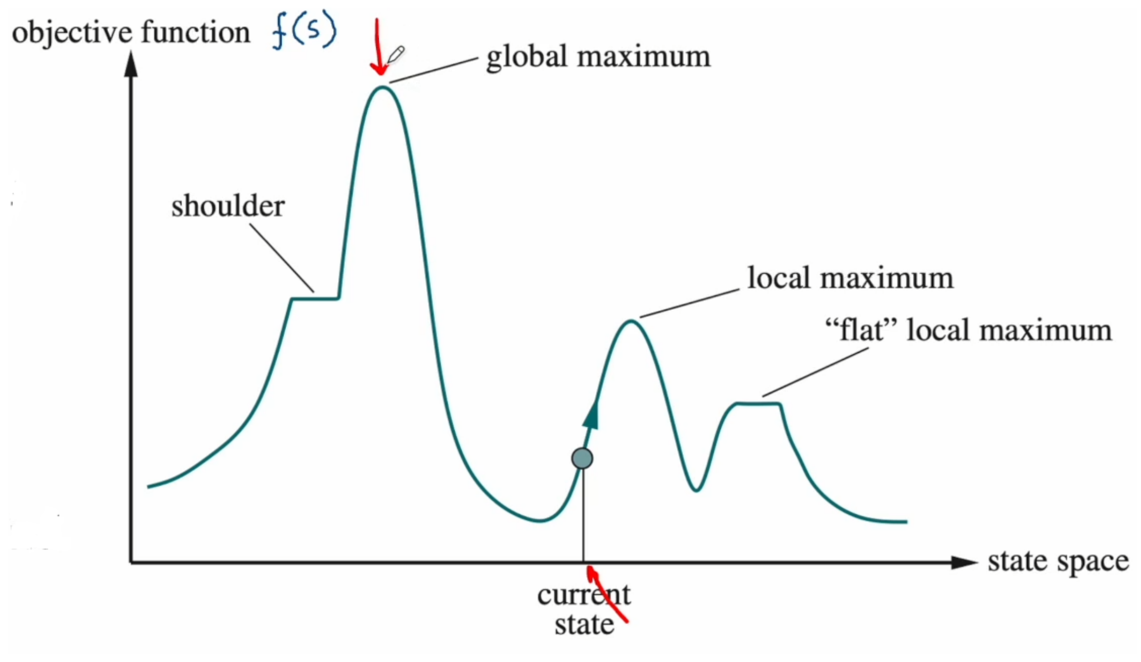
Lecture 5 - Beyond Classical Search

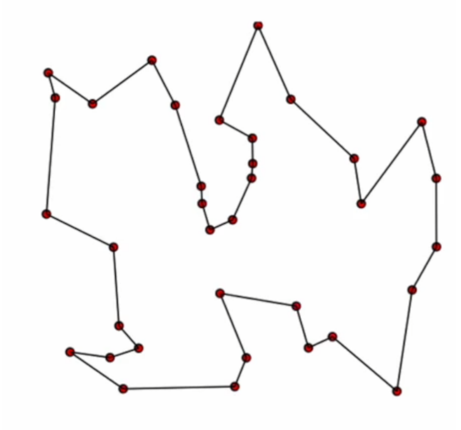
In previous search algorithms, we wanted to find a path from start to goal. This is somewhat of a local approach. In some cases, the path is irrelevant. The goal is to reach the *best* state. The best state is a state with the highest score according to some given function, $f(s)$. This is our objective function, given the state s.



This is an optimization problem. Each state s has a score $f(s)$, and the goal is to find the state with the highest score, or a reasonably high score (This is the difference between a global maximum and a local maximum). Let us get to the **Hill Climbing Algorithm**.

If the elevation corresponds to the cost, then the objective is to find the lowest valley, done through gradient descent. We start with any random start state, we pick the best beighboring state and replace the current state with that one, and loop.

This uses little space, since we only save the current state in memroy. It can find reasonable solutions in large, continuous state spaces where other algorithms are not suitable. An example of this would be the traveling salesman problem.



We want to visit each city exactly once, and return to the starting city. Here, the states are the **orderings** of the cities. The objective function would be the length of the route traveled. The algorithm would go as follows:

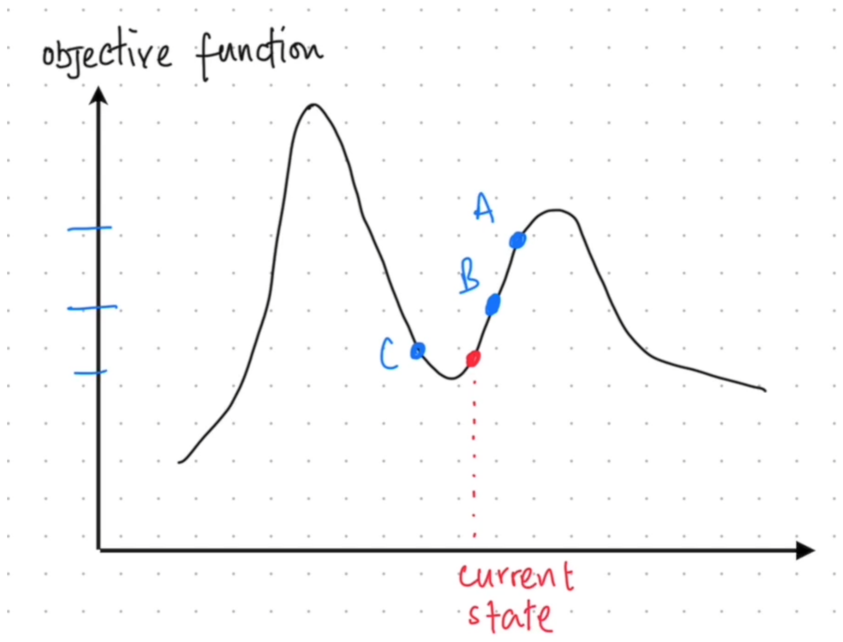
```
function HillClimbing(cities):# Returns solution state
    init_state = {A -> B -> C -> ... } # some random ordering of the cities
    while stopping criterion not met:
        successor = {...} # Some modification to the original ordering
        calculate_f(init_state) # Determine the best state based on the score

return best_state
```

Another example of this would be the 8 queens problem. Now, the states would be any random ordering of the 8 queens. The objective function here would be the number of queens that are **not** attacking each other. Hill Climing can get stuck in local optima:

- Local maximum: A peak that is lower than the highest peak, so it would return a sub-optimal solution.
- Plateau: the valuation function is flat.
- Ridges: Slopes move very gently towrad a peak, so the search may oscillate from side to side.

Stochastic Hill Climbing: We select the uphill moves randomly instead of always choosing the steepest one. The likelihood of selecting a move is proportional to its steepness. By adding this randomness, we can potentially escape local optima and explore more of the state space.



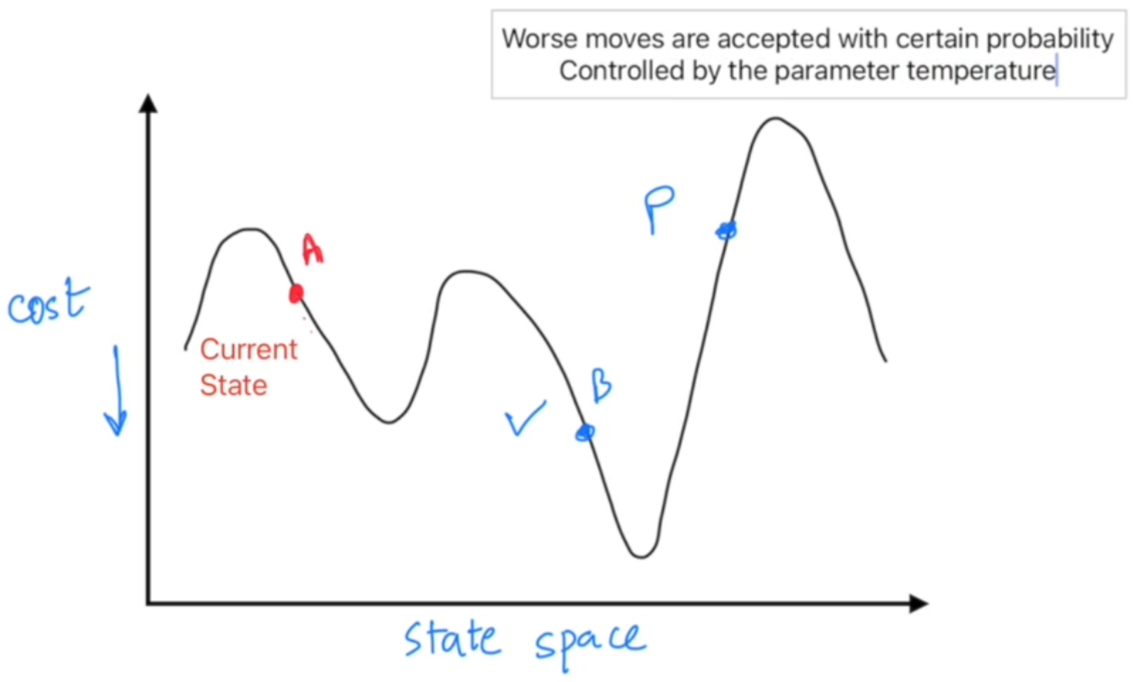
In this example, assume that $f(A)=50$, $f(B)=40$, and $f(C)=10$. Then, to get the probabilities of choosing the three points from where we currently are is given by:

$$\begin{matrix} A \\ B \\ C \end{matrix} \longrightarrow \frac{1}{100}\begin{bmatrix} 50 \\ 40 \\ 10 \end{bmatrix} \longrightarrow \begin{bmatrix} P(A)=0.5 \\ P(B)=0.4 \\ P(C)=0.1 \end{bmatrix}$$

So we have a 10% chance of selecting the point C after the current node, and so on and so forth.

Random Restart Hill Climbing: We restart from a random initial state whenever we get stuck in a local optima. By doing this, we encourage exploring different regions of the state space, increasing chances of finding global maxima.

Simulated Annealing: Mimics annealing process by combining hill climbing and random walk. Random walk enables escape from local minima. When we go from the current state to another, we *could* go to a worse state — it is accepted with a probability controlled by the T parameter (temperature). Demonstrated below:


Worse moves are accepted with certain probability
Controlled by the parameter temperature
cost
A
Current State
P
B
✓
state space

The pseudo-code for simulated annealing:

```
function SIMULATED-ANNEALING(problem, schedule) returns solution state
  current ← problem.initial
  for $t = 1$ to $\infty$ do:
        $T \leftarrow$ schedule($t$)
        if $T = 0$ then:
          return current
        next ← a randomly selected successor of current
        $\Delta E \leftarrow$ value(current) − value(next)
        if $\Delta E > 0$ then:
          current ← next
        else
          current ← next only with probability $\exp\left(\frac{\Delta E}{T}\right)$
```

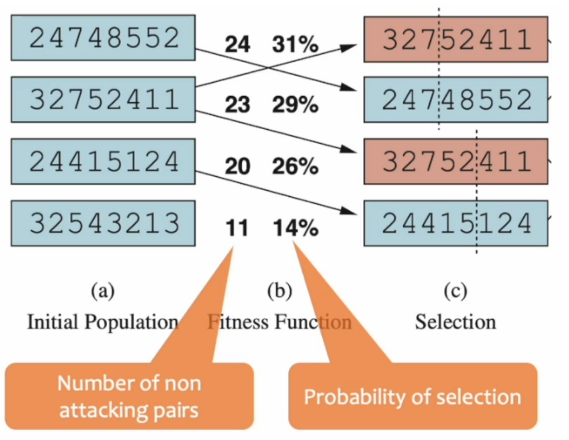
At higher temperatures, the probability of accepting a worse state is higher. As the temperature decreases, the probability of accepting worse moves also decreases, which leads to more exploration of better solutions.

Local Beam Search: Instead of maintaining just one state, we maintain k states in parallel. This is known as the beam. We generate successors of all k states, and we select the best k successors for the next generation until a goal is reached (or stopping condition).

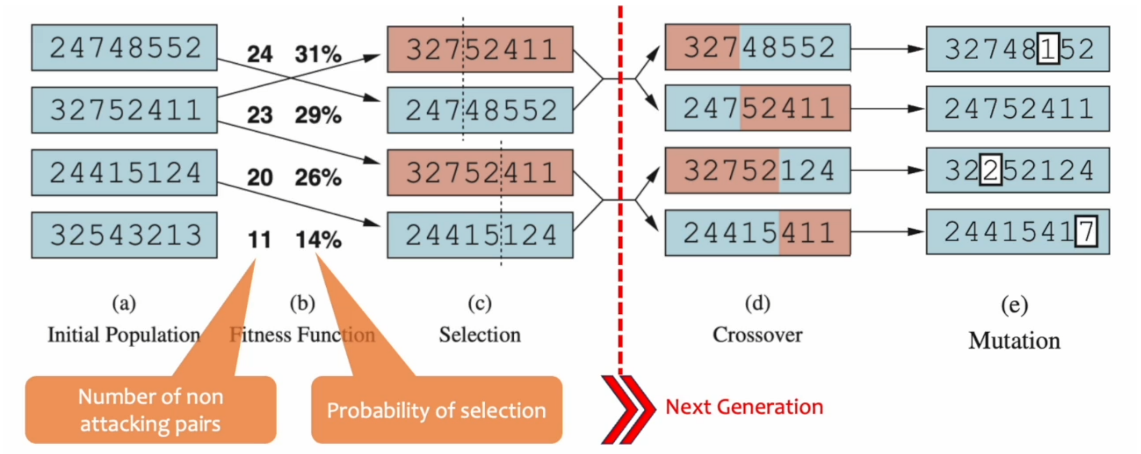
This is good because now we are exploring multiple regions of the state space in parallel.

Genetic Algorithms: Like the beam seach, we have a set of states here as well. This is known as the population here. Each state s is called an individual, often coded up as a string. The objective function here $f(s)$ is called hte fitness of s. We want to find the state s that is the fittest.

Genetic algorithms get the next generation of states based on natural selection, cross-over and mutation. Let's look at the 8 queens problem again to demonstrate how genetic algorithms work. We will use the number of non-attacking pairs as the fitness function $f(s)$. In the worst case, all queens would be attacking all other queens, which would be $8 \times 7/2 = 28$.



In the first one, we have 24 out of 28 non-attacking pairs, which is the best one. We get the probability by summing and normalizing. This is the natural selection process. The next thing is cross-over, where we randomly cross over the strings in the next generation. What about mutation? We randomly select one character in the string and change them.



Now, let's look at the algorithm:

```
GeneticAlgorithm(population, fitness): # returns an individual (one state)
    repeat:
        weights = weighted_by(population, fitness)
        population2 = {}
        for i=1 to size(population) do:
            parent1, parent2 = weighted_random_choices(population, weights, 2)
            child = reproduce(parent1, parent2)
            if (small_random_probability):
                child = mutate(child)
            add child to population2
        population = population2

reproduce(parent1, parent2) # returns an individual
n = length(parent1)
c = random(1, n)
```

```
    return append(substring(parent1, 1, c), substring(parent2, c+1, n))
```

We choose the children based on the weights assigned to the parents, then sometimes we mutate that child. We keep adding these children to the new population and update until we get a good enough fitness value. That is what we return.