Evolutionary Learning

Unit 7

Machine Learning

University of Vienna

Evolutionary Learning

- The Genetic Algorithm (GA)
- The knapsack problem
- The Map Colouring
- The Four Peaks Problem
- The Population-Based Incremental Learning (PBIL)

The Genetic Algorithm

- The genetic algorithm models the genetic process that gives rise to evolution.
- It models sexual reproduction, where both parents give some information to their offspring. Each parent passes on one chromosome out of their two and there is a 50% chance of any gene making it into the offspring.
- There are also random mutations, caused by coping errors when the chromosome material is reproduced, which mean that some thing do change over time.
- Evolution works on a population throught an imaginary „fitness landscape“.

The Genetic Algorithm

- a method for representing problems as chromosomes
- a way to calculate the fitness of a solution
- a selection method to choose parents
- a way to generate offspring by breeling the parents

GA use a **string**, with each element of the string (equivalent to the gene) being chosen from some **alphabet**.

The different values in the alphabet are analogous to the alleles. We have to work out a way of encoding the description of a solution as a string. Then we create a set of random strings to be our initial population.

0-1 Knapsack Problem

- we have a knapsack that has a specific volume
- we have a series of objects to place in it
- each object has a volume and a value
- our goal is best utilize the space in the knapsack by maximizing the value of the objects placed in it.
  An object can be just included zero or one times.
  - The **naive** solution:
    Try every possible combination of L objects and check each of 2^L element afterwards.
  - The **Greedy** Algorithm:
    At each stage it takes the largest thing that hasn't been packed yet and that will still fit into the bag, and iterates that rule.
    This will not necessarily return the optimal solution.

0-1 Knapsack Problem

- This is an NP (Non-deterministic Polynomial time)-complete problem
- It runs in exponential time in the number of objects
- Find the optimal solution for interesting cases is computationally impossible
- We are going to find approximations to the correct solution using a Genetic Algorithm

The knapsack problem: value = volume

- The alphabet is binary and the string is L units long.
- Encode a solution using:
  - 0 for the objects we will not take
  - 1 for the objects we will.
    Create a set (100) of random binary strings of length L by using the random number generator.

    pop = random.rand(popSize, stringLength)
    pop = where(pop $<$ 0.5,0,1)

Fitness

Fitness :

- is the sum of the values of the objects to be taken if they fit into the knapsack but
- if they do not, we will subtract twice the amount by which they are too big for the knapsack from the size of the knapsack

```
fitness = sum(sizes*pop,axis=1)
fitness =
where(fitness>maxSize,500 - 2*(fitness-maxSize),fitness)
```

Generating Offspring: Parent Selection

Following natural selection, fitness will improve if we select strings that are already relatively fit compared to the other members of the population

Basic idea: We choose string α proportionally to its fitness with the probability:

1. $p_\alpha = \frac{F^\alpha}{\sum_{\alpha'} F^{\alpha'}}$, if $F^\alpha \geq 0$ or
2. $p_\alpha = \frac{\exp(sF^\alpha)}{\sum_{\alpha'} \exp(sF^{\alpha'})}$ (Boltzmann selection)

- Scale fitness by total fitness

  fitness = fitness/sum(fitness)
  fitness = 10*fitness/fitness.max()

Fitness Proportional Parents Selection

We will use the Kronecker product kron() to repeat copies of each string in according to scaled fitness from 1 to 10.

- Ignore strings with very low fitness

```
j =0
while round(fitness[j])<1:
j = j+1
newPop = kron(ones((round(fitness[j]),1)),pop[j,:])
```

- Add multiple copies of strings into the newPop

```
for i in range(j+1,self.populationSize):
if round(fitness[i])>=1:
newPop = concatena-
te((newPop,kron(ones((round(fitness[i]),1)),pop[i,:])),axis=0)
```

- Shuffle the order

```
indices = range(shape(newPopulation)[0])
random.shuffle(indices)
newPop = newPop[indices[:self.populationSize],:]
```

Genetic Operators: Crossover

How to combine two strings of breedling pairs from population to generate the offspring?

- **Single point crossover**.
  - We generate the new string as part of the first parent and part of the second.
  - We pic one point at random on the string, and to use parent 1 for the first part of the string, up to crossover point and parent 2 for the rest.
  - We actually generate two offspring, with the second consisting of the first part of parent 2 and the second part of parent 1.
- **Multi-point crossover**.
- The extreme version: **uniform crossover** consist of independently selecting each element of the string at random from the two parents.

Genetic Operators: Crossover

The hope is that sometimes we will take good parts of both solutions and put them together to make an even better solution.

- Single point crossover

```
newPop = zeros(shape(pop))
crossoverPoint =
random.randint(0,self.stringLength,self.popSize)
for i in range(0,self.popSize,2):
    newPop[i,:crossoverPoint[i]] = pop[i,:crossoverPoint[i]]
    newPop[i+1,:crossoverPoint[i]] = pop[i+1,:crossoverPoint[i]]
    newPop[i,crossoverPoint[i]:] = pop[i+1,crossoverPoint[i]:]
    newPop[i+1,crossoverPoint[i]:] = pop[i,crossoverPoint[i]:]
return newPop
```

Genetic Operators: Mutation

- The mutation operator effectively performs local random search.
- The value of each element of the string is changed with some (usually low) probability p.
- For binary alphabet mutation causes a bit-flip: 0 changes to 1 and 1 to 0.
- For chromosomes with real values, some random number is generally added or substracted from the current value.
- Often $p \approx 1/L$ where L is the string length, so that there is approximately one mutation in each string.
- The mutation rate has to trade off doing lots of local search with the risk of disrupting the good solutions.

```
whereMutate = random.rand(shape(pop)[0],shape(pop)[1])
pop[where(whereMutate < self.mutationProb)] = 1 -
pop[where(whereMutate < self.mutationProb)]
```

Elitism

Problem: the best fitness can decrease in the next generation.

This happens because the best strings in the current population are not copied into the next generation, and sometimes none of their offspring are as good.

1. **Elitism:** copy the best strings in the current population into the next population without any change.

```
best = argsort(fitness)
best = squeeze(oldPopulation[best[-self.nElite:],:])
indices = range(shape(population)[0])
random.shuffle(indices)
population = population[indices,:]
population[0:self.nElite,:] = best
```

2. **Tournament:** choose the two fittest out of the two parents and their two offsprings and put them into the new population.

Niching

- Tournaments and elitism reduce the amount of diversity in the population by allowing the same individuals to remain over many generations and encourage **premature convergence** where the algorithm settles down to a constant population that never changes.
  The solutions:
  1. **Niching** (oder **island population**)
     - the population is separated into several subpopulations,
     - a subpopulations evolve independently for some period of time,
     - they are likely to have converged to different local maxima,
     - a few members of one subpopulation are occasionally injectes into another subpopulation.
  2. **Fitness sharing**:
     - the fitness of particular string is averaged across the number of times that that string appears in the population.
     - this biases the fitness function towards uncommon strings
     - but very common gut solutions are selected against.

The Basis Genetic Algorithm

1. **Initialisation**
   - generate N random strings of length L with our choosen alphabet
2. **Learning**
   - repeat:
     * create an (initially empty) new population
     * repeat:
       - select two strings from current population by fitness
       - recombine them to produce two new strings
       - mutate the offspring
       - either add the two offspring to the population or use elitism or tournament
       - keep track of the best string in the population
     * until N strings for the new population are generated
     * replace the current population with the new population
   - until stopping criteria met

Map Colouring

We want to color a graph using k colours in such a way that adjacent regions have different colours. Any two-dimensional planar graph can be coloured with four colours. We want to solve the **three**-colour problem using a GA.

1. Encode possible solution as strings

   Alphabet consists of the three possible shades:

   - black (b),
   - dark (d),
   - light (l)

   For a six-region map possible string is, for example, $\alpha = \{bdblbb\}$

2. Choose a suitable fitness function.

   Count the number of correct edges.

3. Choose suitable genetic operators

   Crossover and mutation.

The Knapsack Problem

1. 20 different packages,
2. a total size of 2436.77
3. a maximum knapsack size of 500:

   sizes = array([109.60,125.48,52.16,195.55,58.67,61.87,92.95,
   93.14,155.05,110.89,13.34,132.49,194.03,121.29,179.33,139.02,
   198.78,192.57,81.66,128.90])

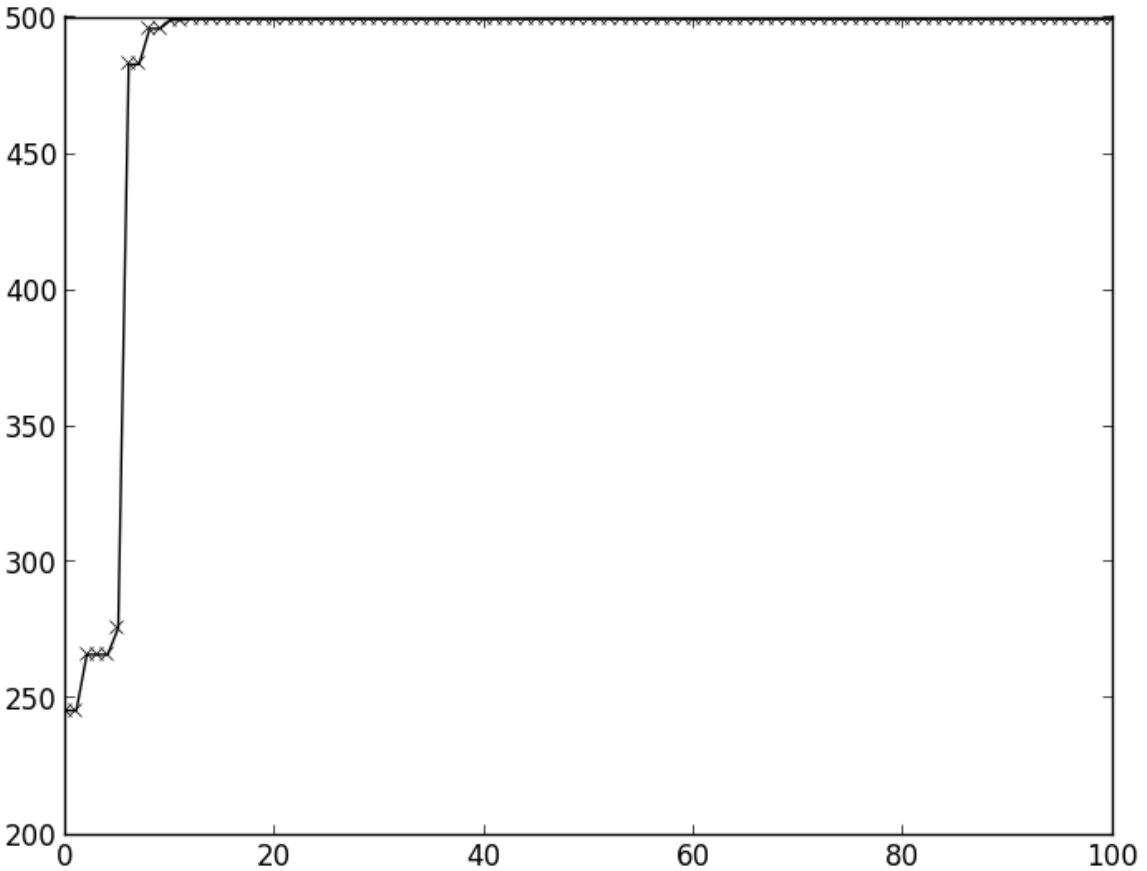
   i. The **exhaustive** search solution:
   [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0.]
   Size = 155.05 + 13.34 + 192.57 + 139.02 = 499.98

   ii. The **greedy** solution:
   Size = 198.78 + 195.55 + 93.14 = 487.47

   iii. The solution of **GA**:
   Size = 499.66

The Knapsack Problem

The GA rapidly finds a near-optimal solution (of 499.66), although in this run it did not find the global optimum.



The Four Peaks Problem

The Four Peaks Problem a very simple **toy problem** and is good for testing algorithms.

The fitness:

- consists of counting the number of 0s at the start, and the number of 1s at the end and returning the maximum,
- if both the number of 0s and the number of 1s are above some threshold value T then the fitness function gets a bonus of 100 added to it.

This is where the name „four peaks“ comes from:

there are two small peaks where there are lots of 0s, or lots of 1s, and then there are two larger peaks, where the bonus is included.

The Four Peaks Problem

- T = 15

  start = zeros((shape(population)[0],1))

  finish = zeros((shape(population)[0],1))

  fitness = zeros((shape(population)[0],1))

- for i in range(shape(population)[0]):

  s = where(population[i,:]==1)

  f = where(population[i,:]==0)

- if size(s)>0: start = s[0][0]

  else: start = 0

- if size(f)>0: finish = shape(population)[1] - f[-1][-1] -1

  else: finish = 0

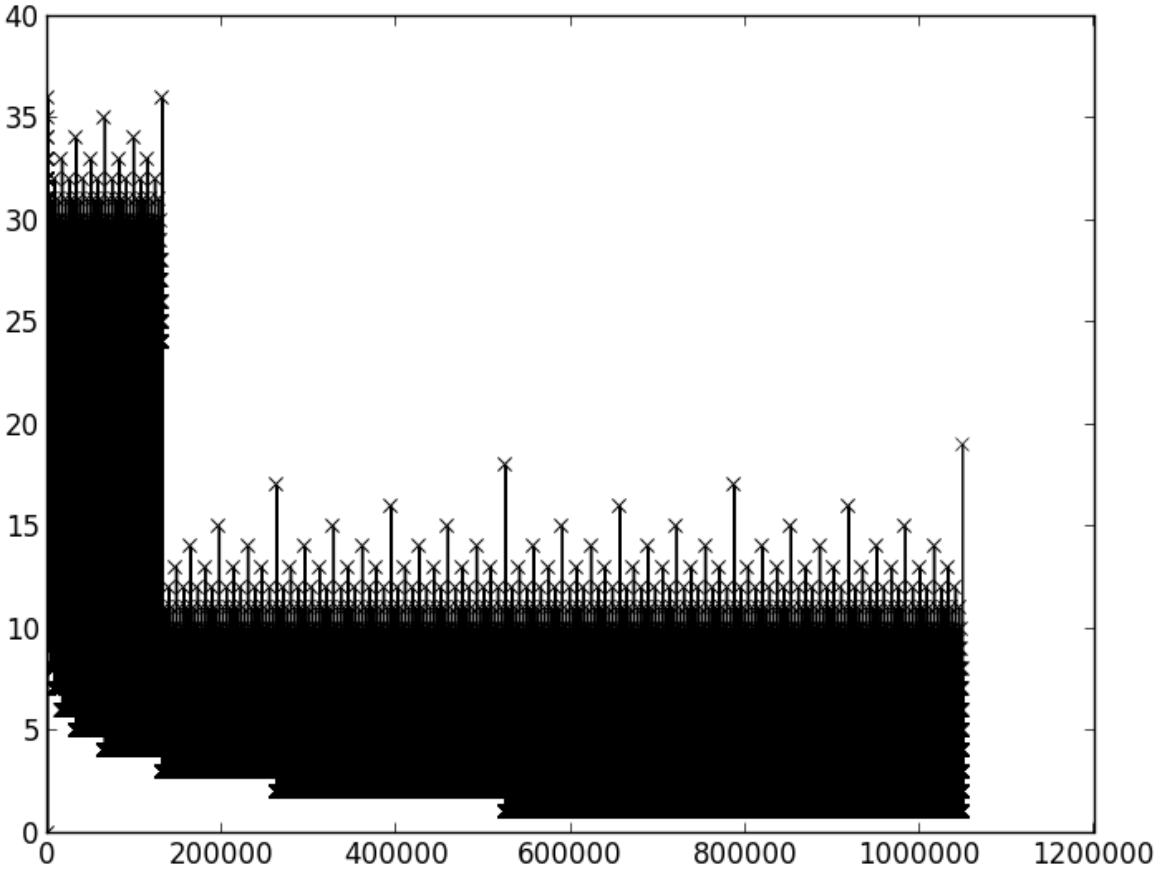
- if start>T and finish>T: fitness[i] = maximum(start,finish)+100

  else: fitness[i] = maximum(start,finish)

The Four Peaks Problem

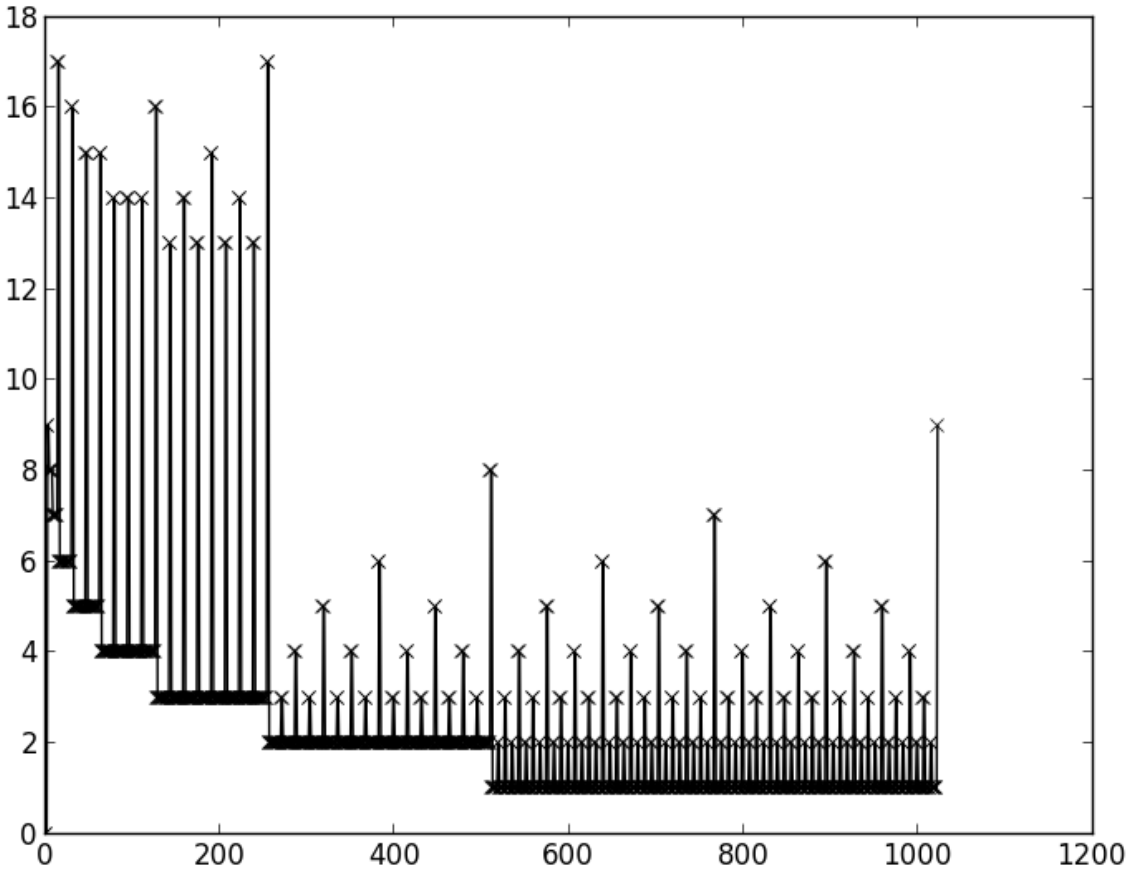
The sampling for poputation with Length = 20, T= 3, Bonus = 20

peaks problem.png



The Four Peaks Problem

The sampling for poputation with Length = 10, T= 2, Bonus = 10



The Four Peaks Problem: Solution I

A chromosome with length 100, $T = 15$, bonus 100.

GA with a mutation rate of 0.01 , which is $1/L$ and single point crossover and an elisitism.

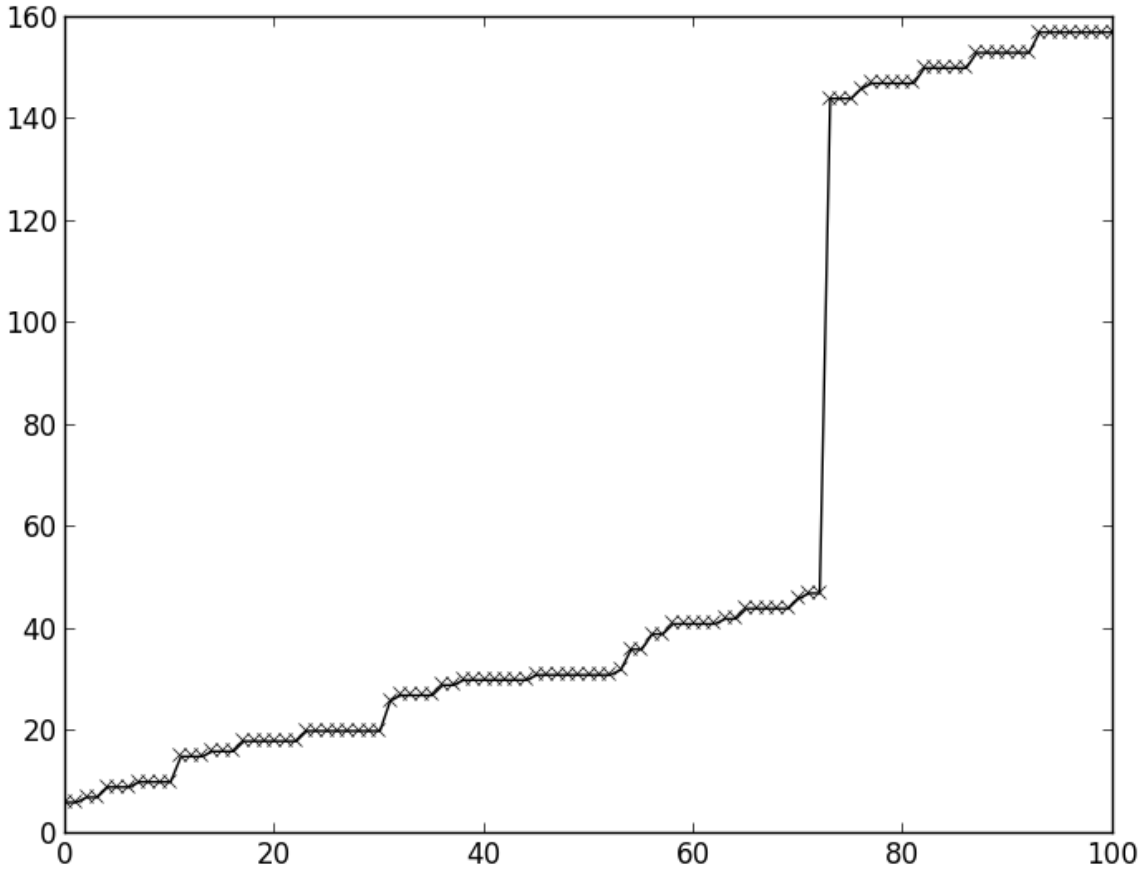
ga = ga(100,'fourpeaks',101,100,0.01,'sp',4, True)

The GA reaches the bonus point.

bestfit= 157.0 = max(57 (0s) + 16 (1s)) + 100 = 157

best population =[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0.
1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1.]

The Four Peaks Problem: Solution I



The Four Peaks Problem: Solution II

The same four peaks problem as before.

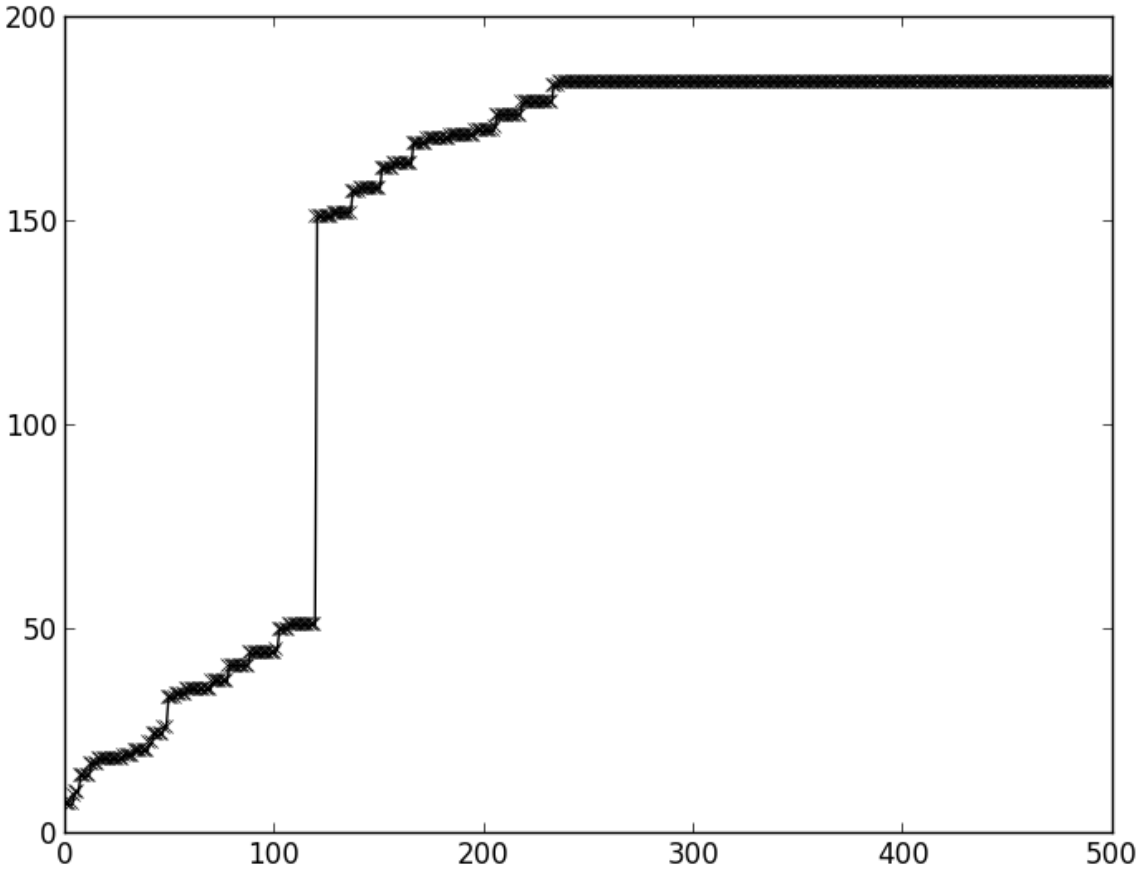
The GA reaches the best possible solution:

the least possible value for 0s and 1s is 16 (must be greater as 15)!

bestfit= 184.0 = max(16 (0s) + 84 (1s)) + 100 = 184

best population =[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

The Four Peaks Problem



Punctuated Equilibrium

- An argument against evolution: the lack of intermediate animals in the fossil record.
- GAs demonstrate one of the explanations why this is not correct,
- The way that evolution actually seems to work is known as **punctuated equilibrium**:
  - There is basically a steady population of some species for a long time
  - Then something changes
  - Over a very short period, there is a big change
  - Then everything settles down again.
- So the chance of finding fossils from the intermediary stage is quite small.

Limitation of the GA

- GA can be very slow
- Once a local maximum has been reached, it can often be a long time before a string is produced that escape from the local maximum
- It is basically impossible to analyse the behaviour of the GA
- We cannot guarantee that the algorithm will converge at all and certainly not to the optimal solution
- You don't know how cautiously you should treat the results
- GAs are usually treated as a black box- strings are pushed in one end and eventually an answer emerges
- It is not possible to improve the algorithm

GAs are widely used when other methods do not work!

Population-Based Incremental Learning (PBIL)

- It works on a binary alphabet like the basic GA
- Instead of maintaining a population, it keeps a probability vector p that gives the probability of each element being 0 or 1.
- Initially, each value of this vector is 0.5, so that each element has equal chance of being 0 or 1.
- A population is then constructes by sampling from the distribution specified vector.
- The fitness of each member of population is computed.
- The two fittest vetors are choosen to update the probability vector, using a learning rate η, which is often set to 0.005

$$p = p \times (1 - \eta) + \eta(\text{best} + \text{second})/2$$

- The population is then thrown away, and a new one sampled from the updated probability vector.

Population-Based Incremental Learning (PBIL)

- Pick best

  best[count] = max(fitness)

  bestplace = argmax(fitness)

  fitness[bestplace] = 0

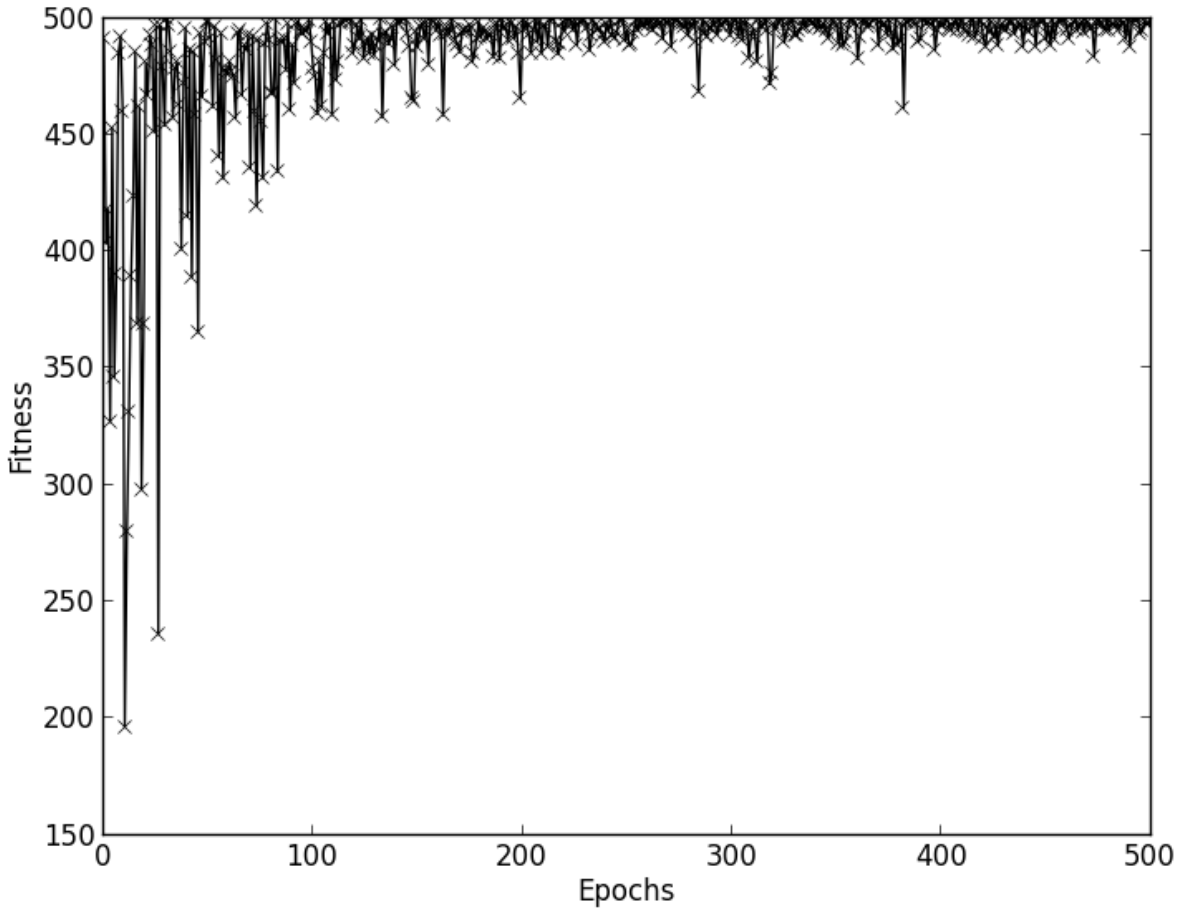
  secondplace = argmax(fitness)

- Update vector

  p = p*(1-eta) +
  eta*((population[bestplace,:]+population[secondplace,:])/2)

The PBIL and the Knapsack Problem

The solution is the same as in the exhaustive search: best= 499.98



The PBIL and the Four Peaks Problem

The best possible solution was found:

best= 176.0

bestPop = [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

The PBIL and the Four Peaks Problem