Algorithm Theory
Spring 2017

Genetic Algorithms

#L13
Date: 24/03/2017

Antorweep Chakravorty

CIPSI
Universitetet i Stavanger

Optimization problems

- Objective function
  - Minimize or maximize
- Can be converted from one form to another

$$min_x f(x) <=> max_x[-f(x)] \qquad max_x f(x) <=> min_x[-f(x)]$$

- A minimization “objective” function is also termed as a “cost” function
- A maximization “objective” function is also termed as a “fitness” function

*Objective functions for any problems are always subjected to certain *constraints*

Hill Climbing (1/3)

- Acts a basic benchmarking *evolutionary algorithm* for optimization problem
- Objective: to get to the highest point in a landscape
  - Take a step in the direction of steepest ascent
  - Re-evaluate the slope of the hill after each step
  - Repeat until there are no directions which leads to a higher slope
- A local search strategy, therefore might not reach convergence

Hill Climbing (2/3)

- Hill Climbing Algorithms
  - Steepest Ascent
    - Conservative algorithm
    - Changes only one solution feature at a time
    - Replaces the current best with the best one-feature changed solution
  - Next Ascent Hill Climbing
    - Like steepest ascent
    - Changes one solution feature at a time
    - More greedy, as soon as a better solution is found, the current solution is replaced
  - **Random Mutation**
    - Like next ascent hill climbing
    - Changes only one solution feature at a time
    - Greedy choices
    - Mutation feature is chooses randomly
  - Adaptive Hill Climbing
    - Like random mutation
    - Generates multiple initial start locations
    - Results of hill climbing depends strongly on initial start conditions
      - to reach a global optimal

Hill Climbing (3/3)

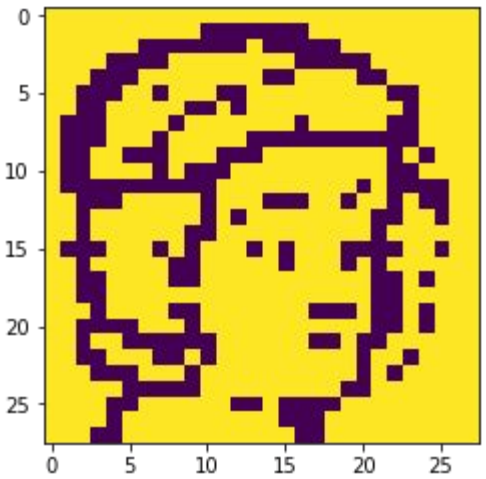


image dim (28, 28)
image pixels 784

```python
def fitness(target, x):
    #We can imagine our image matrix as a flatted vector.
    #Therefore euclidean distance measure, can be used to determine the closeness with another vector(child)
    return np.sqrt(np.sum((target-x)**2))
```

```python
#Random Mutation Hill Climbing Algorithm
start_time = time.time()
np.random.seed(123456789)
n = len(imgV)
#randomly ganerated indivisual of size having the total number of pixels in our image
x0 = np.random.randint(low=0, high=2, size=n, dtype='l')
#keeps track of number of generations
numGen = 0
#while termination critaria not met
while(True):
    showVectorImage(np.reshape(x0, imgDim))
    numGen += 1
    #compute the fitness of x0
    x0_fit = fitness(imgV, x0)

    #If we find a local optimal (usually is the same as global optimal)
    if x0_fit == 0:
        break

    x1 = x0.copy()
    #randomly chosen solution feature 1,n
    q = np.random.randint(low=0, high=n, size=1, dtype='l')
    #replace the q-th solution feature of x1 with a random mutation
    x1[q] = 1 - x0[q]#v
    #Compute the fitness of f(x1) of x1
    x1_fit = fitness(imgV, x1)

    #if the fitness of the child is smaller than the parent (minimization)
    if x1_fit < x0_fit:
        x0 = x1

    print("gen", numGen, "fitness: parent", x0_fit, "child", x1_fit)

print("total gen", numGen)
print("--- %s seconds ---" % (time.time() - start_time))
```

Genetic Algorithms (GA) (1/8)

- Basic features of natural selection
  - A biological system includes a population of individuals, many of which have the ability to reproduce
  - The individuals have a finite life span
  - There is variation in the population
  - Ability to survive is positively correlated to the ability to reproduce
- GA solutions are based on the above principles. Given an optimization problem
  - A population of random candidate solutions are created called *individuals*
  - *Individual's* ability to reproduce and survive are determined based on their *fitness* or closeness as a solution to the objective function
  - *Fit* individuals have a higher chance to reproduce and pass on their genes.
  - Individuals have a *lifespan*, usually until offsprings of their generation are born
  - As generations come and go, the population becomes more fit
  - Under rare circumstances, some individuals in a population *mutate*, i.e. have some genes that change as they get passed on from their parent

Genetic Algorithms (GA) (2/8)

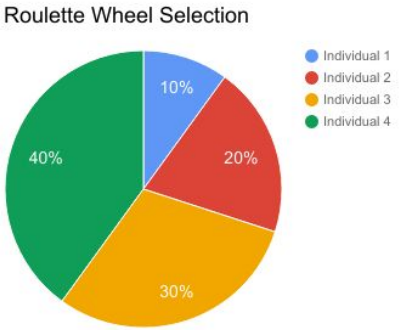
- Individuals are represented using an *encoding scheme* such as a binary/int/char array
  - E.g.: 'xejloxyoklo' => ['x', 'e', 'j', 'l', 'o', 'x', 'y', 'o', 'k', 'l', 'o']
  - Encoding of system parameters is a crucial aspect of GA
  - Has a significant influence on whether the GA really works
  - Encoding are very specific to the given problem
- Fitness
  - Given an individual, fitness determines how close it is to the objective function
  - Eg.: objective create string ‘hello world’ or ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
  - fitness('xejloxyoklo') = 5 and fitness(‘abcdefghijk’) = 0, using fitness function

```
def fitness(target, x):
    fit = 0
    for i in range(0, len(target)):
        if target[i] == x[i]:
            fit +=1
    return fit
```

Genetic Algorithms (GA) (3/8)

- Selection
  - GAs typically start with a population of individuals
  - In each generation fittest individuals in a population have better chance of being selected for reproduction
  - Traditional GA methods
    - Creates the next generations having the same population size as the original
    - Two individuals are selected to be breed and pass on their genes a.k.a *crossover*
    - A pair of parents produces one or more offsprings
    - The genetic structure or the number of genes in the children are the same as their parent
    - The parent generation are killed once the population size is replaced by their children

Genetic Algorithms (GA) (




```
def roulette_wheel(x, f):
    f_sum = sum(f)
    r = np.random.uniform(0, f_sum)

    F = f[0]
    k = 0

    while F < r:
        k += 1
        F += f[k]

    return x[k]
```

- Roulette Wheel Selection
  - A common way to select fit parents
  - Also called as fitness-proportional selection
  - or fitness proportionate selection
  - E.g.:
    - Lets there be a population of size 4
    - Suppose that the individual fitnesses are evaluated as
      - Individual 1: Fitness = 10
      - Individual 2: Fitness = 20
      - Individual 3: Fitness = 30
      - Individual 4: Fitness = 40
    - This would give individual 1 a chance of 10% of being selected to be breed. Individual 2 would have 20%, 30 % for individual 3 and 40% for individual 4.
    - In this example, we select 2 parent to create 2 offsprings in order to keep the population size constant. Each child has a part of their genes from both the parent a.k.a *crossover*
    - We continue selecting parents and creating children until we reach the population limit
    - Therefore, the first spin of the *roulette wheel* will select 2 parent and generate 2 children
    - A second spin would again select 2 parent and generate 2 children
    - Since we reached the population limit, we end this generation by terminating all the parents and proceed with the new generation

Genetic Algorithms (GA) (5/8)

- Crossover
  - A child is created by mixing the genetic information of its parents
  - Multiple methods:
    - Single point crossover
    - Alternating edge crossover
    - Chunk crossover
  - E.g.: using single point crossover
    - Parent 1: 'h', 'e', 'k', 'k', 'm', 'k', 'w', 'o', 'r', 'l', 'x'
    - Parent 2: 'x', 'e', 'j', 'l', 'o', 'x', 'y', 'o', 'k', 'l', 'o'
    - k = 5, using all indexes before this from the 1st parent and the rest from the 2nd parent into the child
      - Child 1: 'h', 'e', 'k', 'k', 'm', 'x', 'y', 'o', 'k', 'l', 'o'
      - Child 2: 'x', 'e', 'j', 'l', 'o', 'k', 'w', 'o', 'r', 'l', 'x'
  -

```
def crossover(x, y):
    n = len(x)
    k = int(n/2)
    c1 = x[:k] + y[k:]
    c2 = y[:k] + x[k:]
    return (c1, c2)
```

Genetic Algorithms (GA) (6/8)

- Mutation
  - Relatively rare, on the order of <= 2%
  - Depends on the problem, population size, encoding and other factors
  - Allows exploration of new potential solution outside the limitation of given population
  - Cases when genetic information crucial to reach convergence is missing from the population
  - GAs typically have a small population size, that could lead to inbreeding and evolutionary dead ends
  - Mutation provides the possibility of injecting that information into the population
- In order to implement mutation, we select a *mutation rate*, typically 1%
- While the crossover process produces a child, each gene of a child would have a probability of 1% to be mutated into something else
- It is important to select a reasonable mutation rate
  - a too high probability would make the GA behave more like a random search
  - A too low mutation probability would result in problems with inbreeding and evolutionary dead ends

```
def mutate(c, p):
    for i in range(0, len(c)):
        r = np.random.uniform()
        if r < p:
            c[i] =
random.choice(string.ascii_lowercase + ' ')
```

Genetic Algorithms (GA) (7/8)

- Objective: Create ordered string
  - "hello world" using random search

```
s = list('hello world')
print(s)
start_time = time.time()
np.random.seed(123456789)
n = len(s)
#population size
N = 100
#Randomly generated indivisuals
x = []
for i in range(0, N):
    x.append(list(''.join(random.choice(string.ascii_lowercase + ' ') for _ in range(n))))
#keeps track of number of generations
numGen = 0
while numGen < 1000:
    numGen += 1
    #stores the fitness of each parent
    f = []
    #calculate the fitness of each parent
    for xi in x:
        f.append(fitness(s, xi))
    bestFit = float("-inf")
    bestFiti = -1
    for i in range(0, N):
        if f[i] > bestFit:
            bestFit = f[i]
            bestFiti = i
    print("gen", numGen, "fitness: (best)", max(f),
          "(avg)", sum(f)/len(f), "(worse)", min(f))
    print("Best Fit", str(x[bestFiti]))
    #Children
    z = []
    while len(z) < len(x):
        x1 = []
        y1 = []
        idx = 0

        while(True):
            idx += 1
            x1 = roulette_wheel(x, f)
            y1 = roulette_wheel(x, f)
            if x1 != y1 or idx == 100:
                break
        c = crossover(x1,y1)
        z.append(c[0])
        z.append(c[1])
    #randomly mutate some of the children
    for c in z:
        mutate(c, 0.01)
    x = z[:]
print("total gen", numGen)
print("--- %s seconds ---" % (time.time() - start_time))
```

Genetic Algorithms (8/8)

- An encoding scheme represents potential solutions to a given problem
- A fitness function that maps problem solutions to fitness values
- Population size
- Selection mothod:
  - Roulette wheel selection
  - Tournament selection
  - Rank selection
  - ...
- Mutation rate: a too high mutation degenerates a GA into random search. A too low mutation rate would not allow the GA to sufficiently explore the search space
- Fitness scaling. Defines how well the fitness function is implemented.
  - A poorly defined fitness function would lead to all individuals having fitness values very close to each other. Such cases would lead to all individuals to be clumped together leading to a poorer selection of parents
  - Alternatively, fitness functions that spreads fitness values much apart might lead to some individuals having almost no chance to be selected
- Crossover type
  - Single point crossover
  - Multiple point crossover
- Specialization/incest. Some GA researchers allow individuals to mate, only if they have common ancestors/similar to each other. Others, on the other hand allow reproduction only if individuals are distinct from each other/belongs to different families/species
- Stopping Criteria.
  - Can run for a predetermined number of generations
  - Can run until the fitness of the best individual is better than some user defined threshold
  - Can run until the fitness of the best individual stops improving over multiple generations

Travelling Salesman Problem (1/5)

- Objective: Finding a minimum distance Hamiltonian path for a given Graph (vertices = cities)
- A NP-Hard Problem
- Time complexity: Exponential of N (the number of cities)
- Simple GA for TSP
  - Step 0:
    - Given a list of cities, each vertex being represented by a coordinate/number
      - *cities = list(np.random.randint(low=0, high=100, size=n))*
    - Encode the solution path as a list of vertices representing the traversed path
    - Distance between cities (edges) can be represented using some distance measure
  - Step 1:
    - Create a population of paths (random forest) connecting each city
      -

```
#Randomly generated paths
x = []
for i in range(0, N):
    p = cities[:]
    np.random.shuffle(p)
    x.append(p)
```

Travelling Salesman Problem (2/5)

- Simple GA for TSP
  - Step 2
    - Determine fitness of each path

```
def fitness(path):
    fit = 0
    for i in range(0, len(path) - 1):
        fit += math.fabs(path[i] - path[i+1])
    #since we have a minimization objective function
    return 1/fit
```

  - Step 3
    - Determine a crossover method
    - Chunk based crossover
      - Randomly determine a chunk size
      - From each parent take an alternating chunk
      - If a chunk creates a cycle,
        - » replace that edge with a random edge
        - » Ensure that the random edge is not present in both the already selected subsequence of genes for the child and in the chunk itself
      - E.g.: with chunk size k = 2
        - » Parent 1: 1345823
        - » Parent 2: 2963578
        - Child 1: 1365827 <- Replaced edges to vertex 3 and 8 from parent 2 with edges to random vertices, as they were creating cycles
        - » Child 2: 2945673 <- Replaced edge to vertex 5 from parent 2

```
def crossover(x, y, cities):
    n = len(x)
    #sub tour length
    k = np.random.randint(low=1, high=n-2)
    c1 = list(x[:k])
    c2 = list(y[:k])

    switch = True
    while True:
        chunk1 = []
        chunk2 = []
        if k >= n:
            break

        if switch:
            chunk2 = x[k:k+k]
            chunk1 = y[k:k+k]
        else:
            chunk2 = y[k:k+k]
            chunk1 = x[k:k+k]


        switch = not switch
        k += k

        remove_circits(chunk1, c1, cities)
        remove_circits(chunk2, c2, cities)

        c1.extend(chunk1)
        c2.extend(chunk2)

    return (c1, c2)
```

```
def remove_circits(chunk, c, cities):
    citiesNotIncluded =
set(cities).difference(c).difference(chunk)
    for i in range(0, len(chunk)):
        if chunk[i] in c:
            choice = np.random.choice(list(citiesNotIncluded),
size=1)
            citiesNotIncluded =
citiesNotIncluded.difference(choice)
            chunk[i] = choice[0]
```

Travelling Salesman Problem (3/5)

- Simple GA for TSP
  - Step 3
    - Use traditional Roulette Wheel selection method to select 2 parents
  - Step 4
    - Create 2 child for each parent
    - Apply mutation rate of 1% or 0.01 for each gene of each child
    - Select two random cities
    - Since each child is a hamiltonian path consisting of all cities, swap the selected cities in the child
    -
```
def mutate(c, p, cities):
    for i in range(0, len(c)):
        r = np.random.uniform()
        if r < p:
            k1 = np.random.randint(low=0, high=len(cities))
            k2 = np.random.randint(low=0, high=len(cities))
            tmp = c[k1]
            c[k1] = c[k2]
            c[k2] = tmp
```
  - Step 5
    - Once the population size of children reaches the one for the current generation, terminate all parents and start with a new generation
  - Step 6
    - Apply any stopping condition as discussed in earlier sections

Travelling Salesman Problem (4/5)

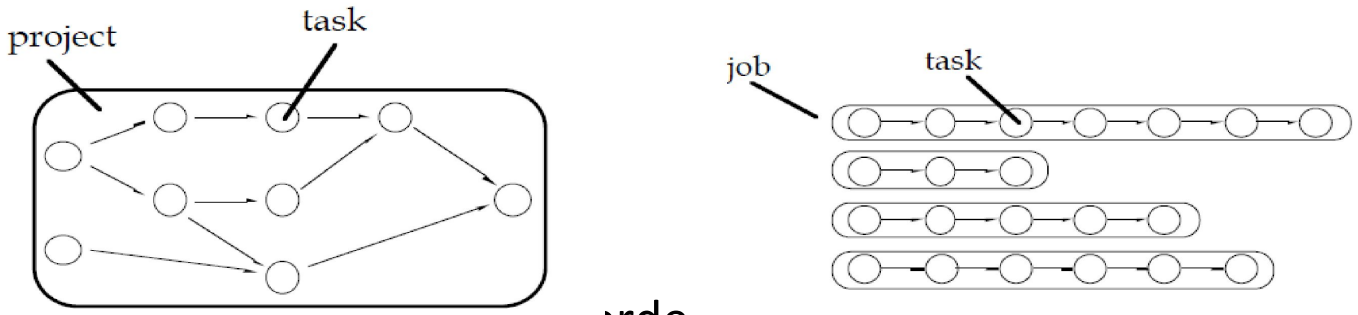
```
start_time = time.time()
np.random.seed(123456789)
#population size
N = 100
#number of cities
n = 10
#cities represented as numbers. Distance between cities is the absolute difference between the integers
#cities = list(np.random.randint(low=1, high=100, size=n))
cities = list(range(1,n))
print("list of cities")
print(cities)
#Randomly generated paths
x = []
for i in range(0, N):
    p = cities[:]
    np.random.shuffle(p)
    x.append(p)
#keeps track of number of generations
numGen = 0
while numGen < 100:
    numGen += 1
    #stores the fitness of each parent
    f = []
    #calculate the fitness of each parent
    for xi in x:
        f.append(fitness(xi))
    bestFit = float("-inf")
    bestFiti = -1
    for i in range(0, N):
        if f[i] > bestFit:
            bestFit = f[i]
            bestFiti = i
    print("gen", numGen, "fitness: (best)", max(f),
          "(avg)", sum(f)/len(f), "(worse)", min(f))
    print("Best Fit", str(x[bestFiti]))
```

Travelling Salesman Problem (5/5)

```
#Children
    z = []
    while len(z) < len(x):
        x1 = []
        y1 = []
        idx = 0

        while(True):
            idx += 1
            x1 = roulette_wheel(x, f)
            y1 = roulette_wheel(x, f)
            if x1 != y1 or idx == 100:
                break

        c = crossover(x1,y1, cities)


        z.append(c[0])
        z.append(c[1])


    #randomly mutate some of the children
    for c in z:
        mutate(c, 0.01, cities)

    x = z[:]


print("total gen", numGen)
print("--- %s seconds ---" % (time.time() - start_time))
```

Project Scheduling versus Job-Shop Scheduling

- **Project scheduling**: a single **project** consists of a set of **tasks** (or **activities**). The tasks have precedence relationships, as shown in the figure below. The tasks may have estimated durations. The main aim of scheduling is to minimize the time to complete the entire project. In multi-project scheduling problems, more than one project must be scheduled.



- **Job shop scheduling**: a work order of ***n* jobs** is to be performed, where each job contains ***m*$_i$ tasks**. Each task has a ***single*** predecessor; each task has its own estimated duration. Objective is to minimize the ***makespan (completion time)*** for the work order.

References

- Evolutionary Optimization Algorithms, DAN SIMON
  - Chapter 2 & 3
- Optional Reading
  - Generic Algorithms for Travelling Salesman Problem, John Grefenstette, et. al.
  - Solving Job-Shop Scheduling Problems by Genetic Algorithm, Mitsuo GEN, et. al.
  - A Genetic Algorithm for Job Shop,Falkenauer, E. and Bouffouix, S.
  - A Genetic Algorithm for Resource-Constrained Scheduling, Wall, PhD Thesis, MIT.