Assegnamento 6 : Evolutionary Computation 1

January 24, 2022

Nicolò Toscani

1 Esercizio 1

Prendendo come riferimento il codice *simple-onemax.py* settare un algoritmo genetico che trova il minimo della funzione

$$f(x, y) = (1.5 + sin(z)) * (sqrt((20 - x)^2 + (30 - y)^2 + 1)$$

dove:

$$x, y, z \in [-250, 250]$$

```
# Specifico se è un problema di minimo settango i pesi
# Pesi positivi: problema di massimo
# Pesi negativi: problema di minimo

# Definisco la funzione obbiettivo. Nel nostro caso una funzione di minimizzazione
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

# Definisco un individuo con un attributo fitness
creator.create("Individual", list, fitness=creator.FitnessMin)

# Lunghezza dei cromosomi
IND_SIZE=3

toolbox = base.Toolbox()

# Attribute generator
#                      define 'attr_bool' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [0,1] (i.e. 0 or 1 with equal
#                      probability)
toolbox.register("attr_float", random.uniform, -250.0, 250.0)

# Structure initializers
#                         define 'individual' to be an individual
#                         consisting of 3 'attr_bool' elements ('genes')
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_float, n = IND_SIZE)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# the goal ('fitness') function to be maximized
def evalOneMax(individual):
    # print("Genotipo individuo: " + str(individual))
    return sum(individual),



def eval_es1_1(individual):
    x = individual[0]
    y = individual[1]
    z = individual[2]
    result = (1.5 + math.sin(z))* (math.sqrt(pow((20-x),2) + pow((30-y),2)) + 1)
    return result,
```

Figure 1: Configurazione algoritmo genetico floating points

```
#----------
# Operator registration
#----------
# register the goal / fitness function
# toolbox.register("evaluate", evalOneMax)
toolbox.register("evaluate", eval_es1_1)

# register the crossover operator, ricombinazione con 2 tagli nel materiale genetico dei genitori
# prendo in input 2 individui e ritono 2 individui ricombinati tra i due genitori
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
# Inverte i valori dei bit del'indivuduo con una probabilità di 0.05 su ogni bit
# toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)

toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.2, indpb=0.05)


# operator for selecting individuals for breeding the next
# generation: each individual of the current generation
# is replaced by the 'fittest' (best) of three individuals
# drawn randomly from the current generation.
toolbox.register("select", tools.selTournament, tournsize=3)

#----------
```

Figure 2: Configurazione algoritmo genetico floating points

Il risultato ottenuto all'iterazione 1000 é:

Best individual is [20.000023524324515, 30.000091097098775, 249.7566132633858], (0.5000470427375316,)

da cui si deduce:

- x: 20.000023524324515
- y: 30.000091097098775
- z: 249.7566132633858
- funzione obiettivo: 0.5000470427375316

Successivamente utilizziamo come rappresentazione del genotipo (codice genetico di un individuo) una lista di 90 interi [0,1].
Il risultato ottenuto all'iterazione 1000 é:

Best individual is [0.1920704785844373, 0.9639863711509189, 0.05504734273120988, 1, -0.03868722987994236, 1.1331578582003403, 0.6115623707116344, -0.2738752642313752, 0.037878196039811955, 0.7505451460953143, 0.6478305992054618, 0.9698427005211111, 0.9380558962933867, 0.8428563046526171, 0.7661226230567094, 1, 1, 1, 1,

```python
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Attribute generator
#                      define 'attr_bool' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [0,1] (i.e. 0 or 1 with equal
#                      probability)
# Come assegnare
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#                         define 'individual' to be an individual
#                         consisting of 100 'attr_bool' elements ('genes')
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 90)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# the goal ('fitness') function to be maximized
def evalOneMax(individual):
    return sum(individual),

def eval_es1_2(individual):
    value = 0
    Min = -250
    Max = 250

    for i in range(30):
        value = value + individual[i] * math.pow(2,i)
    x = Min + value/(math.pow(2,30)-1) * (Max - Min)

    value = 0
    for i in range(30):
        value = value + individual[i+30] * math.pow(2,i)
    y = Min + value/(math.pow(2,30)-1) * (Max - Min)

    value = 0
    for i in range(30):
        value = value + individual[i+60] * math.pow(2,i)
    z = Min + value/(math.pow(2,30)-1) * (Max - Min)

    result = (1.5 + math.sin(z))* (math.sqrt(pow((20-x),2) + pow((30-y),2)) + 1)
    return result,
```

Figure 3: Configurazione algoritmo genetico con lista di bit

-0.03414439608332562, 0.15814734120854904, 0, 0, 1, 0.02031276473494498, 1, 0, 0, 0, 1, 0.24593550207300005, 1.0769991046205516, 1, 1, 0.9412096497083916, 0.16331242628060338, 1.0092571543817002, 1, 0, 0.8391947085276896, -0.10330605004692406, 0.3419689045058007, 0, 1, 0, 1, 0, 0.9453257707962878, 1, 0, 1, 1, 1, 0, 1, 1, 0.09424340168323672, 0, 0, 1, 1, -0.2605038295167279, 1, 1, 0.0044150457623233315, -0.38185675631580973, 0.04429271125161788, 1, 0.8090294908427758, 1, 0, 1.08860763695279, 0.9963998799322066, 0.8564924093788181, 1, 1, 1, 0.3051397859041407, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1.358374770210181, 1, 0], (0.5000000005765113,)

```python
toolbox.register("evaluate", eval_es1_2)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.2, indpb=0.05)

# operator for selecting individuals for breeding the next
# generation: each individual of the current generation
# is replaced by the 'fittest' (best) of three individuals
# drawn randomly from the current generation.
toolbox.register("select", tools.selTournament, tournsize=3)

#----------

def main():
    random.seed(64)

    # create an initial population of 300 individuals (where
    # each individual is a list of integers)
    pop = toolbox.population(n=300)

    # CXPB  is the probability with which two individuals
    #       are crossed
    #
    # MUTPB is the probability for mutating an individual
    CXPB, MUTPB = 0.5, 0.2

    print("Start of evolution")

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit
        print(ind.fitness.values)

    print("ind:" + str(len(ind)))

    print("  Evaluated %i individuals" % len(pop))

    # Extracting all the fitnesses of
    fits = [ind.fitness.values[0] for ind in pop]

    # Variable keeping track of the number of generations
    g = 0

    # Begin the evolution
    while max(fits) > 0.001 and g < 1000:
        # A new generation
        g = g + 1
```

Figure 4: Configurazione algoritmo genetico con lista di bit

da cui si deduce:

- funzione obiettivo: 0.5000000005765113

2 Esercizio 2

L'obbiettivo é quello di indovinare il pattern di 0 e 1 contenuto nel file *smiley.txt*. Si vuole trovare un individuo che corrisponda il piú possibile con quello rappresentato nel pattern fornito.

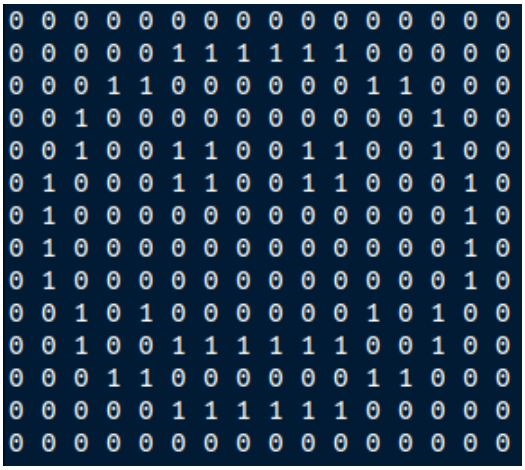

Figure 5: smile 14x16

Di seguito sono riportare alcune prove di esecuzione dell'algoritmo sul file *smiley.txt* tilizzando una popolazione di 300 individui con 1000 iterazioni:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

# Attribute generator
#                      define 'attr_bool' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [0,1] (i.e. 0 or 1 with equal
#                      probability)
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#                         define 'individual' to be an individual
#                         consisting of 100 'attr_bool' elements ('genes')
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 224)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# the goal ('fitness') function to be maximized
def evalOneMax(individual):

    pattern_conv = []

    # Convert individual in pattern format
    for value in pattern:
        pattern_conv.append(value)

    a = np.array(individual)
    b = np.array(pattern_conv)

    result = np.sum(a == b)
    # print(result)

    return result,

#----------
# Operator registration
#----------
# register the goal / fitness function
toolbox.register("evaluate", evalOneMax)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
```

Figure 6: Configurazione algoritmo genetico

Test	**Funzione obiettivo**	**Convergenza**
1	224	741
2	224	818
3	224	1370
4	224	893
5	224	676

Table 1: Esecuzioni smiley.txt con popolazione di 300

Test	**Funzione obiettivo**	**Convergenza**
1	224	1058
2	224	918
3	224	792
4	224	728
5	224	630

Table 2: Esecuzioni smiley.txt con popolazione di 500