Assignment No: 09

[Genetic Algorithm]

CSE 459: Artificial Intelligence

Submitted by

NameAkbar Sharief.
Roll No:AP22110010113
Section: CSE A
Lab Date: 12/3/25
Submission Date: 12/3/25

Department Computer Science and Engineering

School of Engineering and Sciences

SRM University–AP

Amaravati, Andhra Pradesh – 522 240, India

Question

The Knapsack Problem is a classic optimization problem where the objective is to select a subset of items, each with a given weight and value, to maximize the total value while staying within a given weight constraint. Formally, given a set of items with weights wi and values vi, and a knapsack of capacity W, the goal is to find the subset of items that maximizes the total value without exceeding the knapsack capacity.

Genetic Algorithm Parameters:

1. Chromosome Representation: Each chromosome represents a potential solution and is encoded as a binary string of length N, where N is the number of items. The presence of a gene at position i indicates whether the corresponding item is selected (1) or not (0).

2. Initialization: Generate an initial population of potential solutions (chromosomes) randomly

. 3. Fitness Function: Define the fitness function to evaluate the quality of each chromosome. The fitness score is calculated as the total value of the selected items in the knapsack. If the total weight exceeds the knapsack capacity, assign a fitness score of 0.

4. Crossover Policy: Use a two-point crossover policy to create offspring. Randomly select two crossover points on both parent chromosomes and exchange the genetic material between these points to produce two offspring.

5. Mutation Operation: Implement a mutation operation to introduce genetic diversity. Randomly select a gene in a chromosome and flip its value with a certain probability. This simulates the introduction of new genetic material into the population.

6. Selection Strategy: Employ a selection strategy, such as tournament selection, to choose parents for crossover based on their fitness scores. This helps to bias the selection towards fitter individuals.

7. Termination Criteria: Define termination criteria for the genetic algorithm, such as a maximum number of generations or achieving a certain fitness threshold.

Solution

ex4astar.py > ...

```python
import random

cap = 50
w = [10, 20, 30, 40, 50]
v = [60, 100, 120, 150, 200]
ps = 50
gen = 100
mr = 0.1
ts = 5
lim = 10

def gen_ch():
    return [random.choice([0, 1]) for _ in range(len(w))]

def init_pop():
    return [gen_ch() for _ in range(ps)]

def fit(ch):
    tw = sum(c * wi for c, wi in zip(ch, w))
    tv = sum(c * vi for c, vi in zip(ch, v))
    return tv if tw <= cap else 0

def select(pop):
    sel = []
    for _ in range(ps):
        cand = random.sample(pop, ts)
        sel.append(max(cand, key=fit))
    return sel

def cross(p1, p2):
    return [p1[i] if random.random() > 0.5 else p2[i] for i in range(len(p1))]

def mutate(ch):
    pt = random.randint(0, len(ch) - 1)
    ch[pt] = 1 - ch[pt]

def apply_mut(pop):
```

```
36
37  def apply_mut(pop):
38      for ch in pop:
39          if random.random() < mr:
40              mutate(ch)
41
42  def ga():
43      pop = init_pop()
44      best = max(pop, key=fit)
45
46      for _ in range(gen):
47          parents = select(pop)
48          off = [cross(parents[i], parents[i + 1]) for i in range(0, len(parents) - 1, 2)]
49          off = [item for sublist in off for item in sublist]
50          apply_mut(off)
51          pop = off
52          new_best = max(pop, key=fit)
53          if fit(new_best) > fit(best):
54              best = new_best
55
56      return best, fit(best)
57
58  sol, max_v = ga()
59  print("best sol:", sol)
60  print("max value:", max_v)
61
```

Solution:

```
PS C:\Users\mdakb\Desktop\aiml py> & C:/Users/mdakb/AppData/Local/Prog
best sol: [0, 1, 1, 0, 0]
max value: 220
PS C:\Users\mdakb\Desktop\aiml py>
```

Question 2:

Problem Overview: The goal of this problem is to employ a Genetic Algorithm (GA) to evolve a population of strings to match a given target string. The strings are composed of a set of valid genes, and the algorithm iteratively refines the population until an individual perfectly matches the target string. Problem Description: Consider a scenario where we want to generate a specific string using a population-based evolutionary approach. The string is composed of characters from a predefined set of valid genes. The task is to design a Genetic Algorithm to evolve a population of strings, with the aim of reaching the target string. Components of the Problem:

1. Genetic Algorithm: Utilize a population-based approach to evolve strings over generations. Implement functions for initializing the population, calculating fitness scores, selecting parents for crossover, performing crossover, and introducing mutations.
2. Chromosome Representation: Strings serve as chromosomes, where each character is a gene. The set of valid genes includes letters (both uppercase and lowercase), numbers, and special characters.

3. Fitness Function: Define a fitness function that quantifies the similarity between a candidate string and the target string. Fitness is calculated as the count of characters that differ from the corresponding characters in the target string.

4. Crossover Operation: Use a crossover operation to combine genetic information from two parent strings to create offspring. Employ a mechanism to ensure that the crossover respects the composition of valid genes.

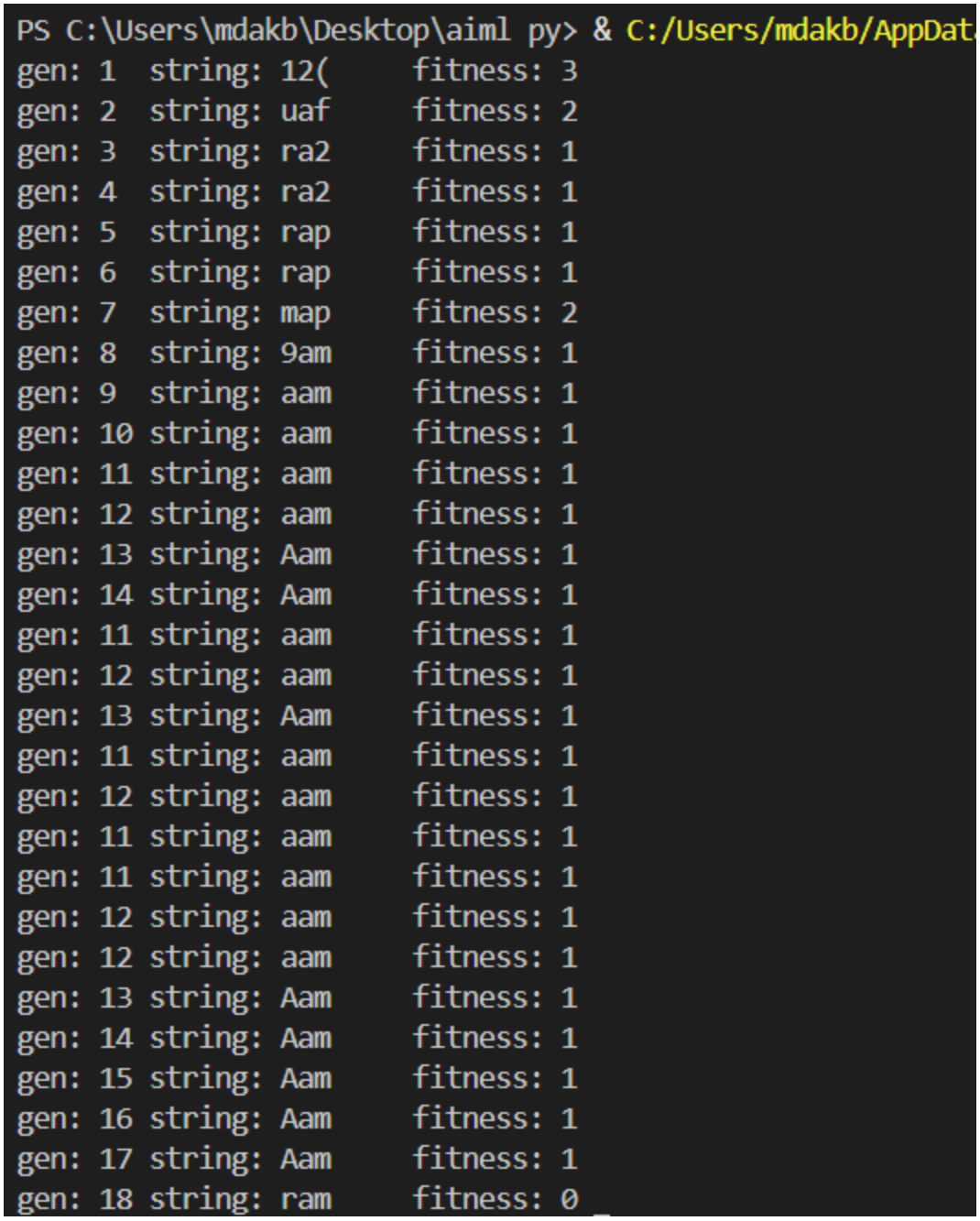
5. Mutation Operation: Implement a mutation operation to introduce small changes in individual strings. The mutation should occur with a low probability and maintain the validity of the genes.

code:

```python
import random

ps = 100
g = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-;:_!\"#%&/()=?@${[]}"
t = "ram"

def r_g():
    return random.choice(g)

def g_gnome():
    return [r_g() for _ in range(len(t))]

def f(c):
    return sum(1 for x, y in zip(c, t) if x != y)

def i_p():
    return [[g_gnome(), 0] for _ in range(ps)]

def s(p):
    return random.sample(p, 2)

def c(p1, p2):
    return [x if (r := random.random()) < 0.45 else y if r < 0.9 else r_g()
            for x, y in zip(p1[0], p2[0])]

def m(c):
    return [r_g() if random.random() < 0.1 else x for x in c]

def ga():
    gen, p = 1, i_p()
    for i in p:
        i[1] = f(i[0])
```

```
        i[1] = f(i[0])

    while True:
        p.sort(key=lambda i: i[1])
        if p[0][1] == 0:
            break

        n_g = p[:ps // 10] + [[c(*s(p)), 0] for _ in range(ps - ps // 10)]
        for i in n_g:
            i[0] = m(i[0])
            i[1] = f(i[0])

        p = n_g
        print(f"gen: {gen}\tstring: {''.join(p[0][0])}\tfitness: {p[0][1]}")
        gen += 1

    print(f"gen: {gen}\tstring: {''.join(p[0][0])}\tfitness: {p[0][1]}")

ga()

```

Solution:



Code Repository:

https://github.com/Akbarpaty/cse455-lab-aiml