Experiment 8

Objective

To be able implement Genetic Algorithm in AI.

Problem Statement

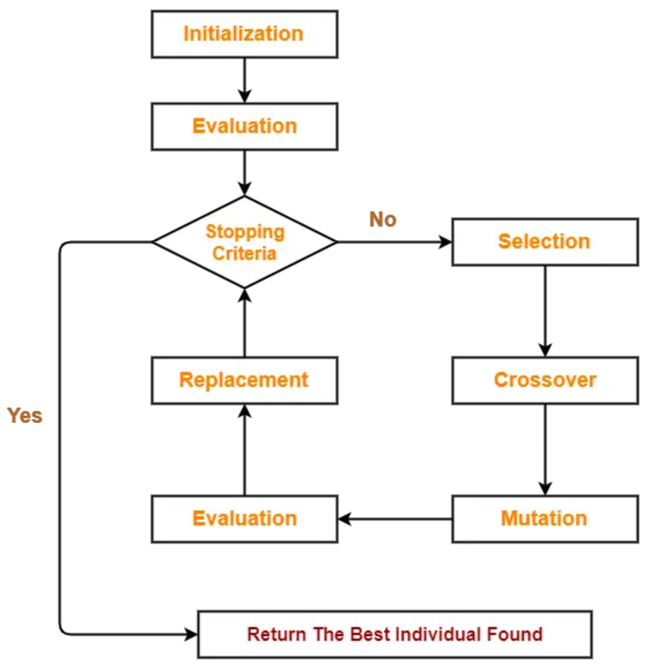
To apply Genetic algorithm for finding out the solution to the problem.

Lab Exercise

Write the program for genetic algorithm, t ooptimize binary strings of fixed length to find the best combination of 0s and 1s that satisfies specific criteria.

Program Code

Example:



How Genetic Algorithm Works

```
'''Genetic Algorithm'''
import random
#Initilize the population

def fitness(parent):
    sum=0
    for i in range(len(parent)):
        sum+=int(parent[i])
    return sum

def initization(population):
    names = list(population.keys())
    n = random.randint(0,3)
    n1 = random.randint(0,3)
    while(n==n1):
        n1 = random.randint(0,3)

    parent1 = names[n]
    parent2 = names[n1]
    return parent1,parent2
```

```python

def crossover(parent1,parent2):
    cp = round(len(parent1)/2)
    n = random.randint(0,1)
    if n==0:
        gen1 = parent1[:cp]+parent2[cp:]
    else:
        gen1 = parent1[cp:]+parent2[:cp]
    return gen1

def mutation(parent1):
    parent1 = list(parent1)
    for j in range(len(parent1)):
        n = random.randint(0,1)
        if n==1:
            if parent1[j]=='0':
                parent1[j]='1'
            else:
                parent1[j]='0'

    par1 = ''.join(parent1)
    return par1

popi = { '1011':0, '1001':0, '1000':0, '1010':0 }

# print(pop)
# print(pop)
# x,y=initization(pop)
# print(x,y)
# x1,y1 = crossover(x,y)
# print(x1,y1)
# x2,y2 = mutation(x1,y1)
# print(x2,y2)

def genetic():
    con = True
    itr = 1
    max_itr = 10000  # Maximum number of iterations
    prev_fitness = None  # Keep track of previous fitness values
    while con and itr <= max_itr:
        popi_copy = popi.copy()
        to_remove = []

        for i in popi_copy:
            par = list(i)
            val = fitness(par)
            popi[i] = val

        x, y = initization(popi)
        x1 = crossover(x, y)
        x2 = mutation(x1)

        val = fitness(x2)
        if x2 not in popi:
            popi[x2] = val
            tmp = min(popi.values())
            for i in popi_copy:
                if popi[i] == tmp:
                    popi.pop(i)
                    break

        li = list(popi.values())

        if all(x == li[0] for x in li) or li == prev_fitness:
            con = False

        print(f"Itr : {itr}")
        print(popi)
        itr += 1
        prev_fitness = li

    if itr > max_itr:
        print("Maximum iterations reached. Termination condition met.")

genetic()
```

Output

```
PS D:\SEM-5\AI> & C:/Users/ASUS/AppData/Local/Programs/Python/Python39/python.exe d:/SEM-5/AI/
    Genetic_algo.py
Itr : 1
{’1011’: 3, ’1001’: 2, ’1010’: 2, ’1111’: 4}
Itr : 2
{’1011’: 3, ’1010’: 2, ’1111’: 4, ’1110’: 3}
Itr : 3
{’1011’: 3, ’1010’: 2, ’1111’: 4, ’1110’: 3, ’0100’: 1}
Itr : 4
{’1011’: 3, ’1010’: 2, ’1111’: 4, ’1110’: 3, ’0100’: 1}
PS D:\SEM-5\AI> & C:/Users/ASUS/AppData/Local/Programs/Python/Python39/python.exe d:/SEM-5/AI/
    Genetic_algo.py
Itr : 1
{’1011’: 3, ’1001’: 2, ’1010’: 2, ’1110’: 3}
Itr : 2
{’1011’: 3, ’1001’: 2, ’1010’: 2, ’1110’: 3}
PS D:\SEM-5\AI> & C:/Users/ASUS/AppData/Local/Programs/Python/Python39/python.exe d:/SEM-5/AI/
    Genetic_algo.py
Itr : 1
{’1011’: 3, ’1001’: 2, ’1010’: 2, ’0011’: 2}
Itr : 2
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4}
Itr : 3
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1000’: 1}
Itr : 4
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1000’: 1, ’0000’: 0}
Itr : 5
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1000’: 1, ’1001’: 2}
Itr : 6
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1001’: 2, ’1110’: 3}
Itr : 7
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1001’: 2, ’1110’: 3, ’0001’: 1}
Itr : 8
{’1011’: 3, ’1010’: 2, ’0011’: 2, ’1111’: 4, ’1001’: 2, ’1110’: 3, ’0001’: 1}
PS D:\SEM-5\AI> & C:/Users/ASUS/AppData/Local/Programs/Python/Python39/python.exe d:/SEM-5/AI/
    Genetic_algo.py
```

Conclusion

The use of a genetic algorithm to optimise binary strings has shown out to be a successful strategy, in conclusion. The algorithm iteratively looks for the best set of 0s and 1s that satisfies the specified requirements through the processes of initialization, fitness evaluation, selection, crossover, and mutation. The algorithm stops when an ideal solution is discovered due to the termination condition, which may be based on reaching a predetermined number of iterations or reaching a certain fitness level. Overall, the employment of a genetic algorithm in this lab report has illustrated its capacity to successfully and efficiently optimise binary strings, highlighting its potential in resolving optimisation issues in numerous fields.