Page No.:
Date:

Experiment 2: (A)

* Aim:- To implement travelling salesman problem using Genetic Algorithm.

* libraries:- random, defaultdicts, numpy

* Theory:- Genetic algorithms are heuristic search algorithm inspired by the process of that supports evolution of life.

• Standard genetic algorithms are divided into five phases:
1. Creating initial population
2. Calculating fitness
3. Selecting best genes
4. Crossing over
5. Mutating to introduce variations.

* Algorithm:
1. Initialize population randomly.
2. Determine the fitness of the chromosome.
3. Until done repeat:
    1. Select parents.
    2. Perform crossover & mutation
    3. Calculate fitness of new population
    4. Append it to gene pool.

```python
import random
from collections import defaultdict  # dictionary of lists
import numpy as np  # for generating random weights in large graph


# Population Initialization
def initialize_population(nodes, pop_size):
    max_nod_num = max(nodes)
    population = []
    for i in range(pop_size):
        chromosome = []
        # to create a fully connected path
        while len(chromosome) != len(nodes):
            rand_node = np.random.randint(max_nod_num + 1)
            # to prevent repeated additions of nodes in the same chromosome
            if rand_node not in chromosome:
                chromosome.append(rand_node)
        population.append(chromosome)
    return population

# Fitness Function
def cost(graph_edges, chromosome):
    total_cost = 0
    i = 1
    while i < len(chromosome):
        for temp_edge in graph_edges:
            if chromosome[i - 1] == temp_edge[0] and chromosome[i] == temp_edge[1]:
                total_cost = total_cost + temp_edge[2]
        i = i + 1
    for temp_edge in graph_edges:
        if chromosome[0] == temp_edge[0] and chromosome[len(chromosome) - 1] == temp_edge[1]:
            total_cost = total_cost + temp_edge[2]
    return total_cost

# Selecting the Fittest Chromosomes
def select_best(parent_gen, graph_edges, elite_size):
    costs = []
    selected_parent = []
    pop_fitness = []
    for i in range(len(parent_gen)):
        costs.append(cost(graph_edges, parent_gen[i]))
        pop_fitness.append((costs[i], parent_gen[i]))
    # sort according to path costs
    pop_fitness.sort(key=lambda x: x[0])
    # select only top elite_size fittest chromosomes in the population
    for i in range(elite_size):
        selected_parent.append(pop_fitness[i][1])
    return selected_parent, pop_fitness[0][0], selected_parent[0]

# Mating
def breed(parent1, parent2):
    # let's say to breed from two parents (0,1,2,3,4) and (1,3,2,0,4)
    # if we choose parent1(0-2) i.e (0,1,2) then we have to choose (3,4) from parent2
    # i.e. we have to create two children from two parents which are disjoint w.r.t each other
    child = []
    childP1 = []
    childP2 = []

    # select two random numbers between range(0,len(parents)) which are used as index
    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    # define start and end index to select child1 from parent1
    if geneA < geneB:
        startGene, endGene = geneA, geneB
    else:
        endGene, startGene = geneA, geneB

    # add parent1(startGene,endGene) to child1
    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    # add parent2 to child2 if parent2 not in child1
    childP2 = [item for item in parent2 if item not in childP1]

    # create new child using disjoint Child1 and Child2
    child = childP1 + childP2
```

```
        return child

def breedPopulation(parents, pop_size):
    children = []
    temp = np.array(parents)
    n_parents = temp.shape[0]
    # create new population of size pop_size from previous population
    for i in range(pop_size):
        # choose random parents
        random_dad = parents[np.random.randint(low=0, high=n_parents - 1)]
        random_mom = parents[np.random.randint(low=0, high=n_parents - 1)]
        # create child using random parents
        children.append(breed(random_dad, random_mom))
    return children

# Mutation
def mutate(parent, n_mutations):
    # we cannot randomly change a node from chromosome to another node
    # as this will create repeated nodes
    # we define mutation as mutation of edges in the path i.e. swapping of nodes in the
chromosome
    temp_parent = np.array(parent)
    size1 = temp_parent.shape[0]
    max_nod_num = max(parent)
    for i in range(n_mutations):
        # choose random indices to swap nodes in a chromosome
        rand1 = np.random.randint(0, size1)
        rand2 = np.random.randint(0, size1)
        # if rand1 and rand2 are same, then chromosome won't be mutated
        # so change rand2
        if rand1 == rand2:
            rand2 = (rand2 + 1) % size1
        parent[rand1], parent[rand2] = parent[rand2], parent[rand1]
    return parent


def mutatePopulation(population, n_mutations):
    mutatedPop = []
    # mutate population
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], n_mutations)
        mutatedPop.append(mutatedInd)
    return mutatedPop

# Genetic Algorithm implementation

# class that represents a graph
class Graph:
    def __init__(self, vertices):
        self.nodes = []  # list of nodes
        for i in range(len(vertices)):
            self.nodes.append(vertices[i])
        self.edges = []  # to store graph
        # dictionary with the lists of successors of each node, faster to get the successors
        # each item of list is a 2-tuple: (destination, weight)
        self.successors = defaultdict(list)

    # function that adds edges
    def addEdge(self, u, v, w):
        for edges in self.edges:
            # check if edge is already present
            if u == edges[0] and v == edges[1]:
                print("Edge already exists")
                return
        self.edges.append([u, v, w])
        self.successors[u].append((v, w))

    # function to get the cost of optimal path found
    def get_cost(self, visited_nodes):
        if len(visited_nodes) <= 1:
            return 0
        else:
            total_cost = 0
            i = 1
            while i < len(visited_nodes):
                for temp_edge in self.edges:
```

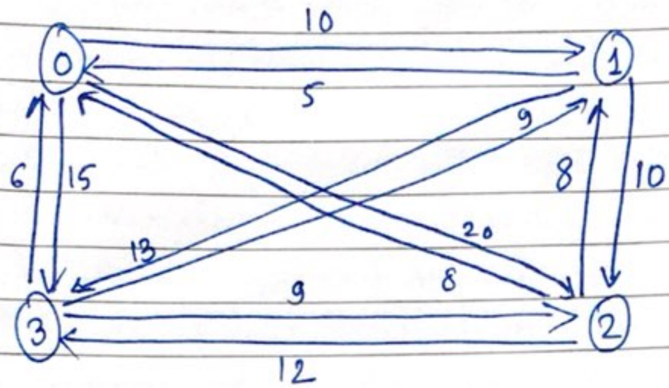
```
temp_edge[1]:          if visited_nodes[i - 1] == temp_edge[0] and visited_nodes[i] ==
                    total_cost = total_cost + temp_edge[2]
                i = i + 1
            for temp_edge in self.edges:
                if visited_nodes[0] == temp_edge[0] and visited_nodes[len(visited_nodes) - 1]
== temp_edge[1]:
                    total_cost = total_cost + temp_edge[2]
            return total_cost

    def disconnected(self, initial_node):
        is_disconnected = False
        for node in range(len(self.nodes)):
            neighbors = self.successors[node]
            # graph is fully connected if number of neighbours of each node will be 1 less
than total
            # number of nodes in the graph
            if len(neighbors) < (len(self.nodes) - 1):
                is_disconnected = True
                return is_disconnected
        return is_disconnected

    def gen_algo(self, source, generations):
        # check if a graph is fully connected
        if self.disconnected(source):
            print("Graph is not connected")
            return []
        # initialize population with a certain size
        pop_size = 20
        parent_gen = initialize_population(self.nodes, pop_size)
        print(parent_gen)
        # keep the track of minimum path cost for each generation
        overall_costs = []
        # keep track of best route with minimum path cost for each generation
        overall_routes = []
        for i in range(generations):
            print("Generation number :", i + 1, "/", generations)
            # choose only elite chromosome from population
            elite_size = 10
            parent_gen, min_cost, best_route = select_best(parent_gen, self.edges, elite_size)
            print("Best route for generation", i + 1, ":", best_route)
            print("Best cost for generation", i + 1, ":", min_cost)
            # store minimum path cost and best route for every generation
            overall_costs.append(min_cost)
            overall_routes.append(best_route)
            # mating
            parent_gen = breedPopulation(parent_gen, pop_size)
            # mutating
            n_mutations = 1
            parent_gen = mutatePopulation(parent_gen, n_mutations)
            print(
"=========================================================================================")
        # select the minimum path cost
        minimum = min(overall_costs)
        min_index = -1
        # find the path with minimum path_cost from stored overall_routes
        # find the path with minimum path_cost from stored overall_routes
        for i in range(len(overall_costs)):
            if minimum == overall_costs[i]:
                min_index = i
        # return best route
        return overall_routes[min_index]


def print_path(path, source):
    print("================Path found================")
    print("final path:")
    start = path.index(source)
    for i in range(start, len(path) - 1):
        print(path[i], "->", path[i + 1])
    print(path[len(path) - 1], "->", path[0])
    for i in range(0, start):
        print(path[i], "->", path[i + 1])


g = Graph([0, 1, 2, 3])
```

```
g.addEdge(0, 1, 10)
g.addEdge(1, 0, 5)
g.addEdge(0, 2, 20)
g.addEdge(2, 0, 8)
g.addEdge(0, 3, 15)
g.addEdge(3, 0, 6)
g.addEdge(1, 2, 10)
g.addEdge(2, 1, 8)
g.addEdge(1, 3, 9)
g.addEdge(3, 1, 13)
g.addEdge(3, 2, 9)
g.addEdge(2, 3, 12)

generations = 3
path = g.gen_algo(0, generations)  # executes the algorithm
total_cost = g.get_cost(path)
if total_cost:
    print_path(path, 0)
    print("total_cost", total_cost)
else:
    print('Did not reach the goal!')
```

OUTPUT:

```
TSP-2
"C:\Users\HP\PycharmProjects\AIMLTC1 LABS\venv\Scripts\python.exe" C:\Users\HP\AppData\Roaming\JetBrains\PyCharmCE2022.2\scratches\TSP-2.py
[[2, 3, 0, 1], [3, 2, 1, 0], [2, 0, 3, 1], [1, 0, 2, 3], [1, 2, 0, 3], [0, 2, 1, 3], [0, 3, 1, 2], [3, 2, 0, 1], [0, 1, 3, 2], [3, 1, 0, 2], [1, 0
Generation number : 1 / 3
Best route for generation 1 : [3, 2, 1, 0]
Best cost for generation 1 : 28
====================================================================================================
Generation number : 2 / 3
Best route for generation 2 : [3, 2, 1, 0]
Best cost for generation 2 : 28
====================================================================================================
Generation number : 3 / 3
Best route for generation 3 : [1, 3, 2, 0]
Best cost for generation 3 : 31
====================================================================================================
=================Path found=================
final path:
0 -> 3
3 -> 2
2 -> 1
1 -> 0
total_cost 28

Process finished with exit code 0
```

* Graph for the given problem:



* Conclusion :- Hence, we have successfully Implemented Travelling salesman problem in python using genetic algorithm.

Page:
Date: / /

Experiment : 2

- Aim:- To implement BFS and DFS Algorithm

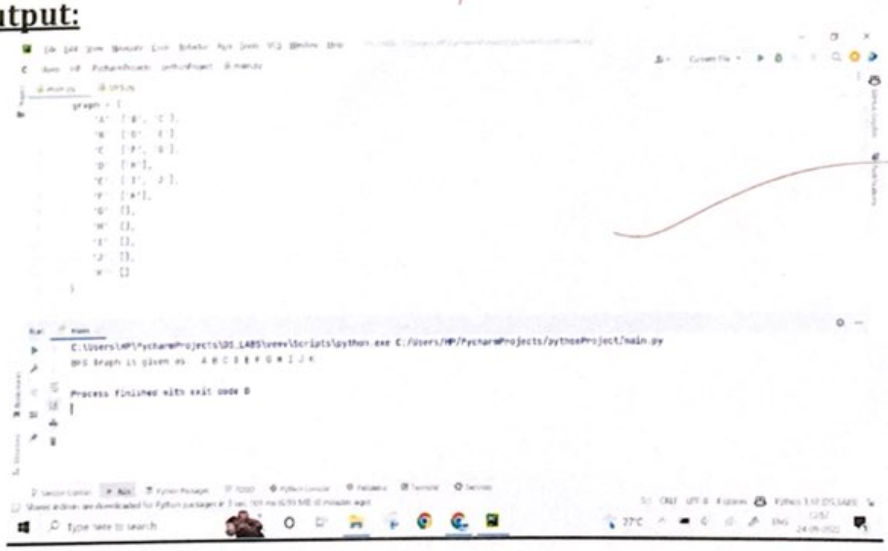
- Theory:- Breadth First Search and Depth First Search in python are algorithms used to traverse a graph or a tree.

1] Breadth First Search:- It follows the process of traversing each node of the graph/tree layer by layer. A standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited ; 2) Not Visited. So the purpose of the algorithm is to visit all the vertex while avoiding cycles.
To implement BFS, we use a queue data structure, to store the intermediate nodes, According to the FIFO (First In First Out) property, everytime we dequeue from the queue, we always get the oldest intermediate node. Then we expand it. And when we visited a node, we indicate it as visited, ~~at~~ and enqueue all its uninvited nodes to the queue. This way we always end up traversing the tree layer-by-layer. If we visit our goal node, we just terminate the searching.

- **BFS Graph Search**
    - **Code:**

```
# TC1_LAB2
# SIA VASHIST  20190802107
# BFS Search & DFS Search


graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H'],
    'E': ['I', 'J'],
    'F': ['K'],
    'G': [],
    'H': [],
    'I': [],
    'J': [],
    'K': []
}


def bfs_connected_component(graph, start):
    # keep track of all visited nodes
    explored = []
    # keep track of nodes to be checked
    queue = [start]

    # keep looping until there are nodes still to be checked
    while queue:
        # pop shallowest node (first node) from queue
        node = queue.pop(0)
        if node not in explored:
            # add node to list of checked nodes
            explored.append(node)
            neighbours = graph[node]

            # add neighbours of node to queue
            for neighbour in neighbours:
                queue.append(neighbour)
    return explored


# drivers code
path = bfs_connected_component(graph, 'A')
print("BFS Graph is given as: ", " ".join(path))
```

- **Output:**

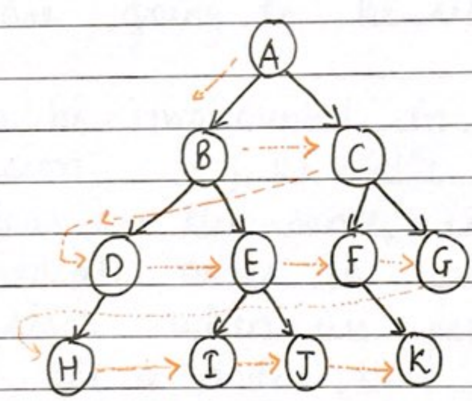


* ~~Algorithm~~:- Pseudocode:-

create a queue Q
mark v as visited and put v into Q
While Q is non-empty
    remove the head u of Q.
    mark and enqueue all unvisited neighbours of Q.

* Time Complexity :- $O(|V| + |E|)$
Space Complexity :- $O(|V| + |E|)$ for ~~adjacency~~ Adjacency List.

* Graph for traversal :-



Page:
Date: / /

2] Depth First Search:- The Depth First Search is a recursive algorithm that uses the concept of backtracking.
Its process involves thorough searches of all nodes by going, ahead if potential, else by backtracking.
Here, the word backtrack basically means once you are moving forward and there are not any more nodes along the present path, you progress backwards on an equivalent path to seek out nodes to traverse.
All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

It can be implemented in two approaches:-
i) recursion ; ii) while loop.
The ideas are the same, insted of storing the intermediate nodes in queue, we push them into stack. When we pop a stack, we always get a node that arrived the latest.

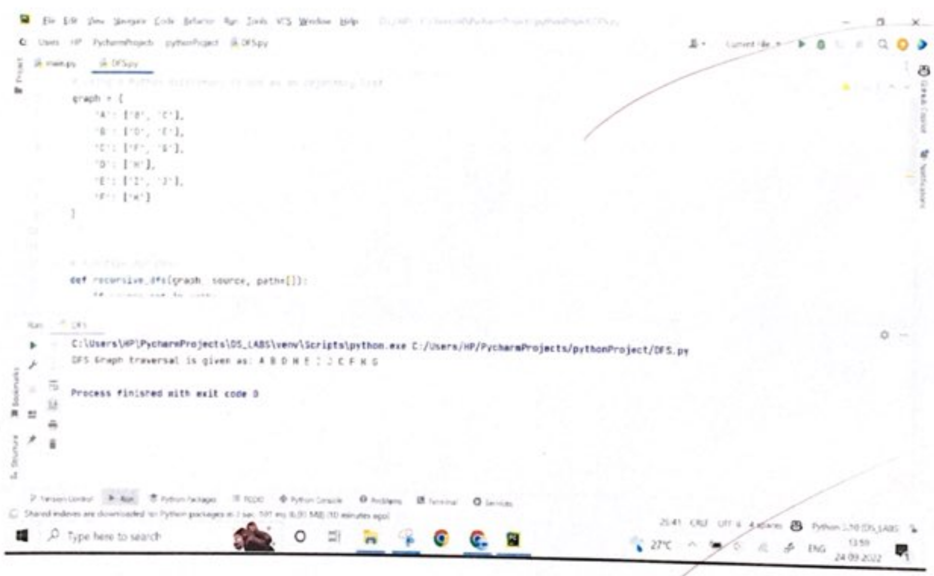
* Complexity :-
b = branching factor ; m = max depth of search tree

- Time complexity :- $O(b^m)$
- Space Complexity :- $O(mb)$

- **DFS Graph Search**
  - **Code:**

```python
# Using a Python dictionary to act as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H'],
    'E': ['I', 'J'],
    'F': ['K']
}


# function for dfs:
def recursive_dfs(graph, source, path=[]):
    if source not in path:
        path.append(source)
        if source not in graph:
            # leaf node, backtrack
            return path
        for neighbour in graph[source]:
            path = recursive_dfs(graph, neighbour, path)
    return path


# Driver Code
path = recursive_dfs(graph, 'A')
print("DFS Graph traversal is given as:", " ".join(path))
```

- **Output:**

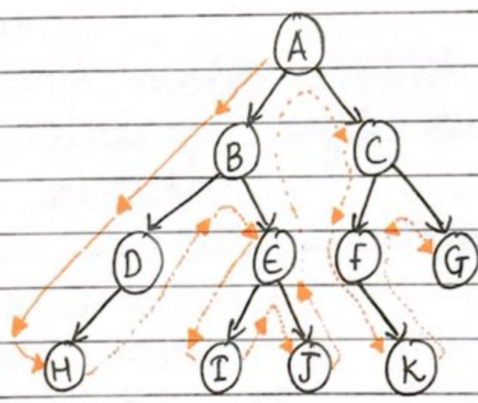


Page:
Date: / /

* Pseudocode :-

```
DFS (G,u)
    u.unvisited = true
    for each v ∈ G.adj[u]
        if v.visited == false
            DFS (G,v)
init(){
    For each u ∈ G
        u.visited = false
    for each u ∈ G
        DFS(G,u)
}
```

* Graph for traversal :-



* Conclusion :- Hence, we have implemented & seen graph traversal technique through BFS & DFS algorithms successfully.

28/03/22