Laboratory Manual

Name of the School:	**School of Computing Sciences and Engineering**
Name of the Department:	**Computer Science and Engineering**
Name of the Programme: **Semester :**	**BTech AIML** **VI**
Course/ Course Code:	**Search Algorithm 17YCM611**
Academic Year:	**2022-23**

Guidelines

Laboratory rules

1. Attendance is required for all lab classes. Students who do not attend lab will not receive credit.
2. Ensure that you are aware of the test and its procedure before each lab class. **You will NOT be allowed to attend the class if you a renort unprepared!**
3. Personal safety is top priority. Do not use equipment that is not assigned to you.
4. All accidents must be reported to your instructor or laboratory supervisor.
5. The surroundings near the equipment must be cleaned before leaving each lab class.
6. Ensure that readings are checked and marked by your TA for each lab period.

Laboratory report

1. Each student has to submit the report for each experiment.
2. Your report is to be written only in the space provided in the manual
3. Please write your number and, batch and group number.
4. Your report must be neat, well organized, and make a professional impact. Label all axes and include proper units.
5. Your reports should be submitted within 7 days and before the beginning of the lab class of the next batch.
6. Your reports will be as per rubrics provided at the end of each experiment

Anyone caught plagiarizing work in the laboratory report, from a current or past student's notebook, will receive 0 on the grading scale.

Name of Student .....

Academic Year .....

Programme .....

Class/ Roll No. .....

PRN No. .....

CERTIFICTAE

This is to certify that,

Mr./Miss …….

PRN No……of class……

has satisfactorily/unsatisfactorily completed the Term Work of

Course ……

in this School during academic year …… .

Course Teacher Head of Department Dean Academics

Table of Contents

Lab : 1

Aim : Perform DFS using python

Traversal means that visiting all the nodes of a graph which can be done through Depth-first search or Breadth-first search in python. Depth-first traversal or Depth-first Search is an algorithm to look at all the vertices of a graph or tree data structure. Here we will study what depth-first search in python is, understand how it works with its bfs algorithm, implementation with python code, and the corresponding output to it.

What is Depth First Search?

What do we do once have to solve a maze? We tend to take a route, keep going until we discover a dead end. When touching the dead end, we again come back and keep coming back till we see a path we didn't attempt before. Take that new route. Once more keep going until we discover a dead end. Take a come back again… This is exactly how Depth-First Search works.

The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward 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.

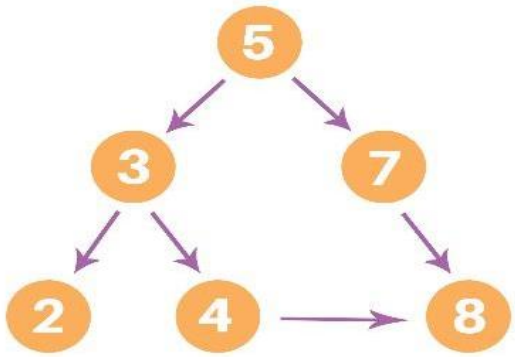
DFS Algorithm

Before learning the python code for Depth-First and its output, let us go through the algorithm it follows for the same. The recursive method of the Depth-First Search algorithm is implemented using stack. A standard Depth-First Search implementation puts every vertex of the graph into one in all 2 categories: 1) Visited 2) Not Visited. The only purpose of this algorithm is to visit all the vertex of the graph avoiding cycles.

The DSF algorithm follows as:

1. We will start by putting any one of the graph's vertex on top of the stack. 2. After that take the top item of the stack and add it to the visited list of the vertex. 3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.

4. Lastly, keep repeating steps 2 and 3 until the stack is empty.



DFS pseudocode

The pseudocode for Depth-First Search in python goes as below: In the init() function, notice that we run the DFS function on every node because many times, a graph may contain two different disconnected part and therefore to make sure that we have visited every vertex, we can also run the DFS algorithm at every node.

```
DFS(G, u)

u.visited = 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)

}
```

DFS Implementation in Python (Source Code)

Now, knowing the algorithm to apply the Depth-First Search implementation in python, we will see how the source code of the program works.

Consider the following graph which is implemented in the code below:

```
# Using a Python dictionary to act as an adjacency list

 graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : ['F'],

'D' : [],

'E' : ['F'],

'F' : []

}

print(graph)

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs
```

```
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, 'A')
```

Output :

```
{'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}
Following is the Depth-First Search
A
B
D
E
F
```

Time Complexity

The time complexity of the Depth-First Search algorithm is represented within the sort of O(V + E), where V is that the number of nodes and E is that the number of edges.
The space complexity of the algorithm is O(V).

Applications

Depth-First Search Algorithm has a wide range of applications for practical purposes. Some of them are as discussed below:

1. For finding the strongly connected components of the graph
2. For finding the path
3. To test if the graph is bipartite
4. For detecting cycles in a graph
5. Topological Sorting
6. Solving the puzzle with only one solution.
7. Network Analysis
8. Mapping Routes
9. Scheduling a problem

Conclusion

Hence, Depth-First Search is used to traverse the graph or tree. By understanding this practical, you will be able to implement Depth-First Search in python for traversing connected components and find the path.

Lab : 2

Aim : Perform BFS Using Python

Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

BFS Algorithm

As breadth-first search is the process of traversing each node of the graph, 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.

BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS uses a queue.

The steps of the algorithm work as follow:

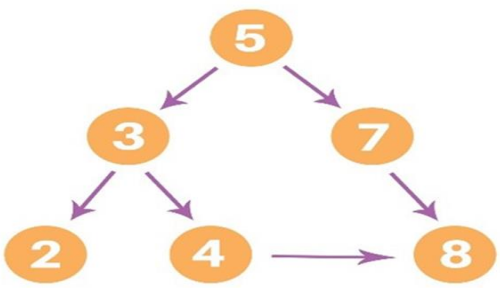
1. Start by putting any one of the graph's vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.

Many times, a graph may contain two different disconnected parts and therefore to make sure that we have visited every vertex, we can also run the BFS algorithm at every node.

BFS pseudocode

The pseudocode for BFS in python goes as below:

1. create a queue Q
2. mark v as visited and put v into Q
3. while Q is non-empty
4. remove the head u of Q
5. mark and enqueue all (unvisited) neighbours of u



Program :

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

OUTPUT :

Following is the Breadth-First Search

5 3 7 2 4 8

Lab : 3

Aim : Determine whether goal- or data-driven search would be preferable for solving each of the following problems. Justify your answer.

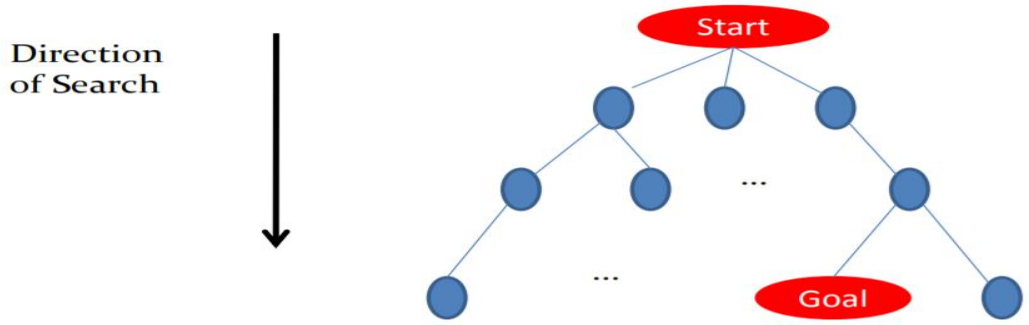
a. Diagnosing mechanical problems in an automobile.

b. You have met a person who claims to be your distant cousin, with a common ancestor named “John Doe”. Verify the claim.

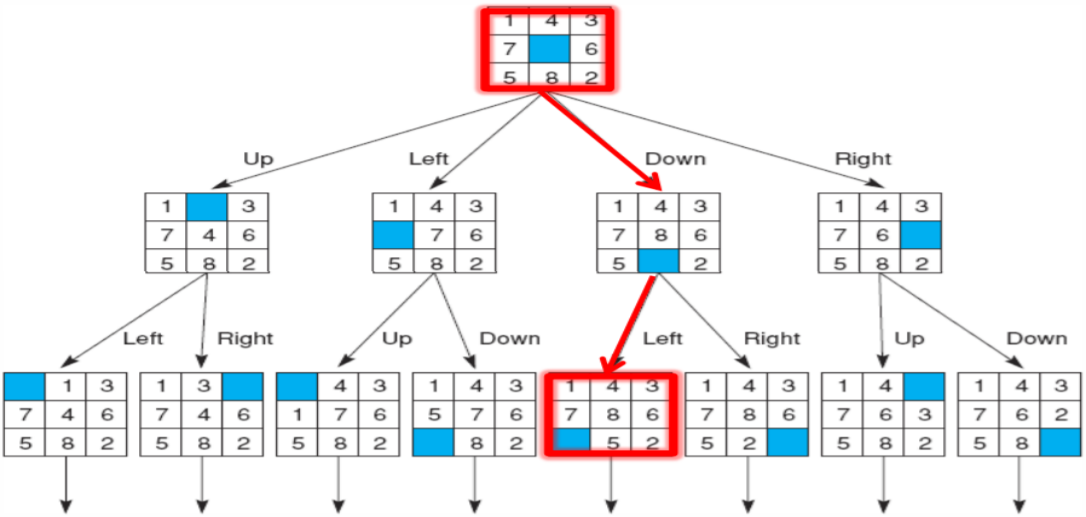
c. Another person claims to be your distant cousin. He does not know the ancestor’s name, but knows that it was more than eight generations back. You would like to find this ancestor or determine she does not exist.

Data-Driven and Goal-Driven Search

So far we have been assuming that – the start state contains the current state of our problem and is at the root of our search and – the goal state is at the fringe.

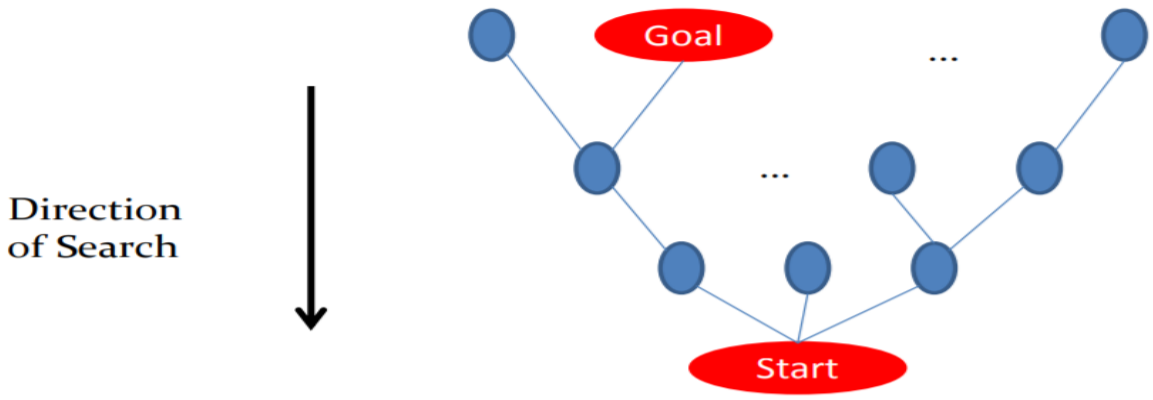


Data-Driven Search Example:



Data-Driven and Goal-Driven Search

Alternatively: we could start with the goal and work our way back to the initial state



Goal-Driven Search

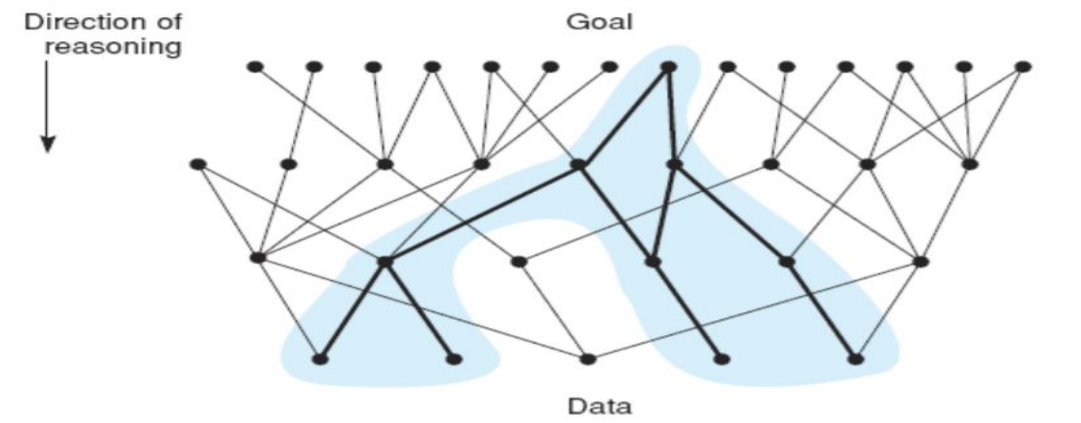
1. Find what states could have produced the goal

2. Consider those states as subgoals

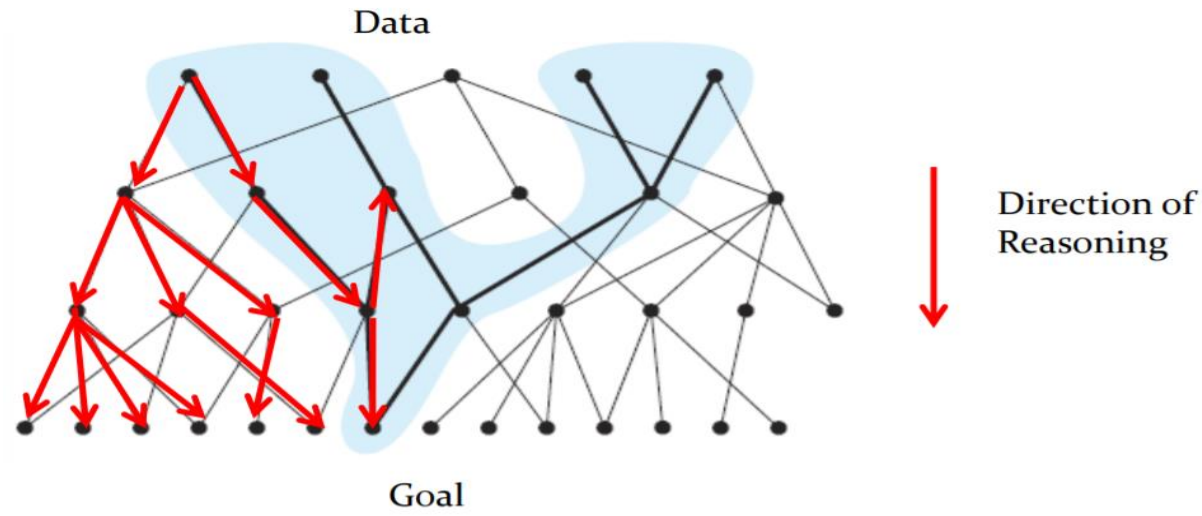
3. Repeat for each subgoal until an initial state is found

Data-Driven or Goal-Driven Search?

Problem Structure		Example	Goal-Driven	Data Driven
Ease of Determining Goal	Goal is given	Theorem Prover	X	
	Difficult to form a goal or hypothesis	DENDRAL expert System		X
Branching Factor	Large number of rules producing many possible goals	Theorem Prover	X	
	Large number of potential goals and few ways to use the information	DENDRAL expert System		X
Availability of Data	Data not given but must be acquired by system	Medical Diagnosis Systems	X	
	Data is given	Geological data analysis by PROSPECTOR		X



State space in which goal-directed search effectively prunes extraneous search paths.



Data-driven search examines at least 13 nodes before reaching the goal

Goal-driven search examines 10 nodes before reaching the goal

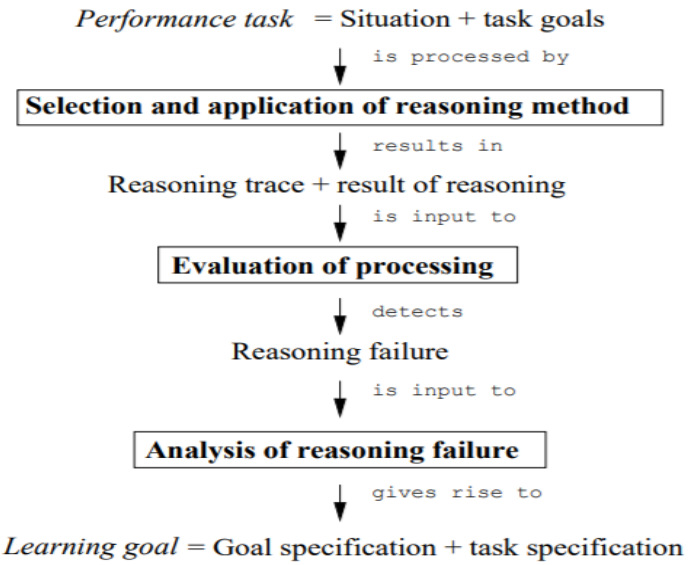


Figure: Generation of learning goals

Note-

There are different types of blind search algorithms.

Which one is more appropriate for the problem on hand depends on:

• The algorithm's characteristics.

• The computing time and memory resources available.

The characteristics of the solution needed:

• Importance of finding the shortest path.

• Whether we want all solutions or a single solution

The problem being solved:

• The size of the search space including its branching factor.

• The depth at which the solution is expected to be found.

Data Driven-

Production set:

1. $p \wedge q \rightarrow goal$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $start \rightarrow v \wedge r \wedge q$

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

Space searched by execution:

Goal Driven Search-

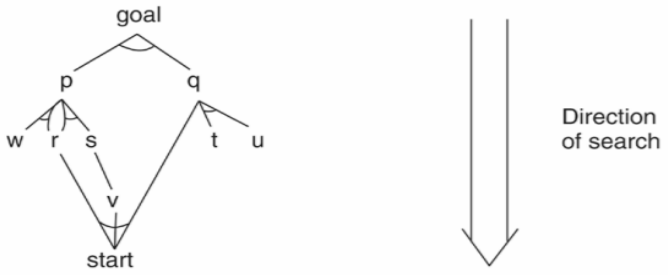
Production set:

1. p ∧ q → goal
2. r ∧ s → p
3. w ∧ r → p
4. t ∧ u → q
5. v → s
6. start → v ∧ r ∧ q

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal, p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

Space searched by execution:



Answer-

a) Usually goal-driven, the goal being the problem. It could also be like EMYCIN, where data is collected and possible goals are suggested data-driven). Then each goal is checked (goal-driven).

Answer

Use a combination of methods - “many frontier search”:

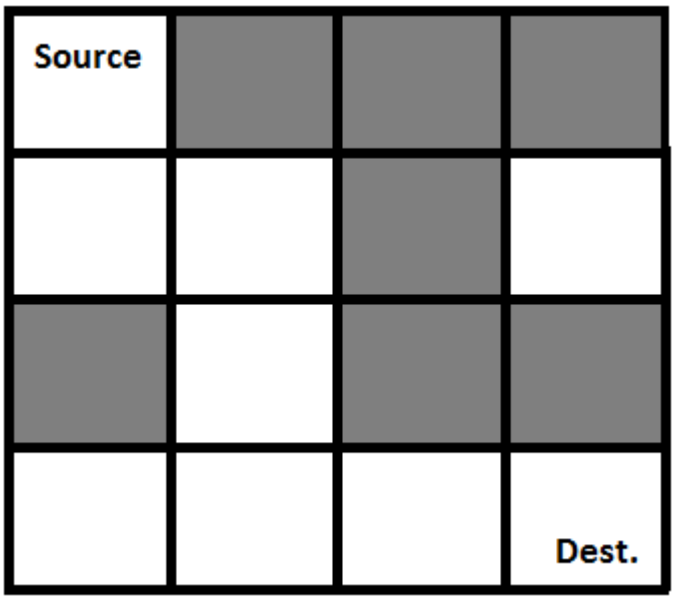
(1) Develop several generations of John Doe’s descendants.

(2) Develop several generations of your ancestors and of the alleged cousin’s ancestors. Look for common vertices in these graphs.

c. Another person claims to be your distant cousin. He does not know the ancestor’s name, but knows that it was more than eight generations back. You would like to find this ancestor or determine she does not exist. Goal-driven, twice: develop ancestor trees for both you and the “cousin”, look for people common to these.

Lab : 4

Aim : Python3 program to solve Snake in a Maze



Program code-

```
# Python3 program to solve Snake in a Maze
# problem using backracking
# Maze size
N = 4

# A utility function to print solution matrix sol
def printSolution( sol ):

    for i in sol:
        for j in i:
            print(str(j) + " ", end="")
        print("")

# A utility function to check if x,y is valid
# index for N*N Maze
def isSafe( maze, x, y ):
```

```
    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True

    return False
""" This function solves the Maze problem using Backtracking.
It mainly uses solveMazeUtil() to solve the problem. It
    returns false if no path is possible, otherwise return
    true and prints the path in the form of 1s. Please note
    that there may be more than one solutions, this function
    prints one of the feasable solutions. """
def solveMaze( maze ):

    # Creating a 4 * 4 2-D list
    sol = [ [ 0 for j in range(4) ] for i in range(4) ]

    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist");
        return False

    printSolution(sol)
    return True

# A recursive utility function to solve Maze problem
def solveMazeUtil(maze, x, y, sol):

    #if (x,y is goal) return True
    if x == N - 1 and y == N - 1:
        sol[x][y] = 1
        return True
```

```
        # Check if maze[x][y] is valid
        if isSafe(maze, x, y) == True:
            # mark x, y as part of solution path
            sol[x][y] = 1

            # Move forward in x direction
            if solveMazeUtil(maze, x + 1, y, sol) == True:
                return True

            # If moving in x direction doesn't give solution
            # then Move down in y direction
            if solveMazeUtil(maze, x, y + 1, sol) == True:
                return True

            # If none of the above movements work then
            # BACKTRACK: unmark x,y as part of solution path
            sol[x][y] = 0
            return False
    # Driver program to test above function
    if __name__ == "__main__":
    # Initialising the maze
        maze = [ [1, 0, 0, 0],
                 [1, 1, 0, 1],
                 [0, 1, 0, 0],
                 [1, 1, 1, 1] ]
        solveMaze(maze)
```

OUTPUT :

```
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```

Lab : 5

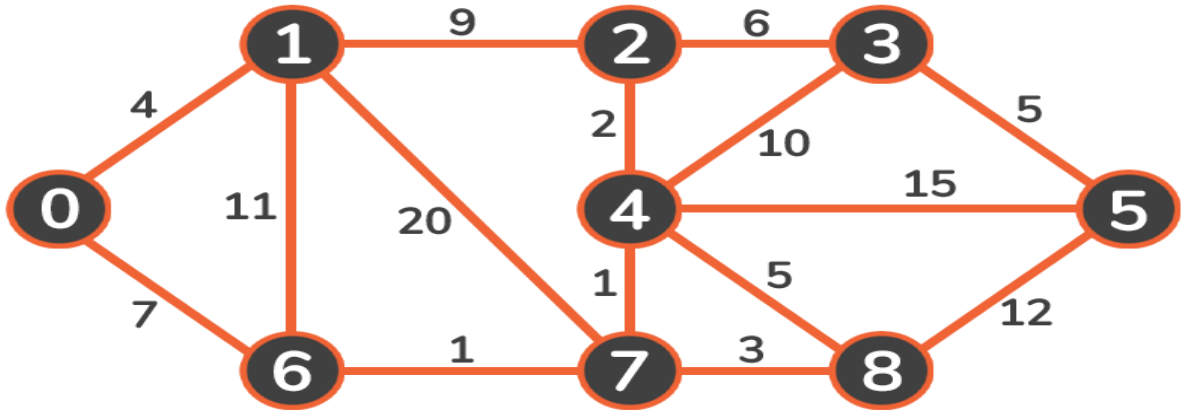
Aim : Given a graph and a source vertex in graph ,find the shortest paths from source to all vertices in the given

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn't include all vertices:

- Pick a vertex u which is not there in *sptSet* and has minimum distance value.
- Include u to *sptSet*.
- Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of a distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.



Let's say that *Vertex 0* is our starting point. We are going to set the initial costs of vertices in this graph to infinity, except for the starting vertex:

vertex	cost to get to it from vertex 0
0	0
1	inf
2	inf
3	inf
4	inf
5	inf
6	inf
7	inf
8	inf

We pick the vertex with a minimum cost - that is *Vertex 0*. We will mark it as visited and add it to our set of visited vertices. The starting node will *always* have the lowest cost so it will always be the first one to be added:

Then, we will update the cost of adjacent vertices (1 and 6). Since 0 + 4 < infinity and 0 + 7 < infinity, we update the costs to these vertices:

vertex	cost to get to it from vertex 0
0	**0**
1	4
2	inf
3	inf
4	inf
5	inf
6	7
7	inf
8	inf

Now we visit the next smallest cost vertex. The weight of *4* is lower than *7* so we traverse to *Vertex 1*:

Upon traversing, we mark it as visited, and then observe and update the adjacent vertices: 2, 6, and 7:

- Since 4 + 9 < infinity, new cost of vertex 2 will be 13
- Since 4 + 11 > 7, the cost of vertex 6 will remain 7
- Since 4 + 20 < infinity, new cost of vertex 7 will be 24

These are our new costs:

vertex	cost to get to it from vertex 0
0	**0**
1	**4**
2	13
3	inf
4	inf
5	inf
6	7
7	24
8	inf

The next vertex we're going to visit is *Vertex 6*. We mark it as visited and update its adjacent vertices' costs:

vertex	cost to get to it from vertex 0
0	**0**
1	**4**
2	13
3	inf
4	inf
5	inf
6	**7**
7	8
8	inf

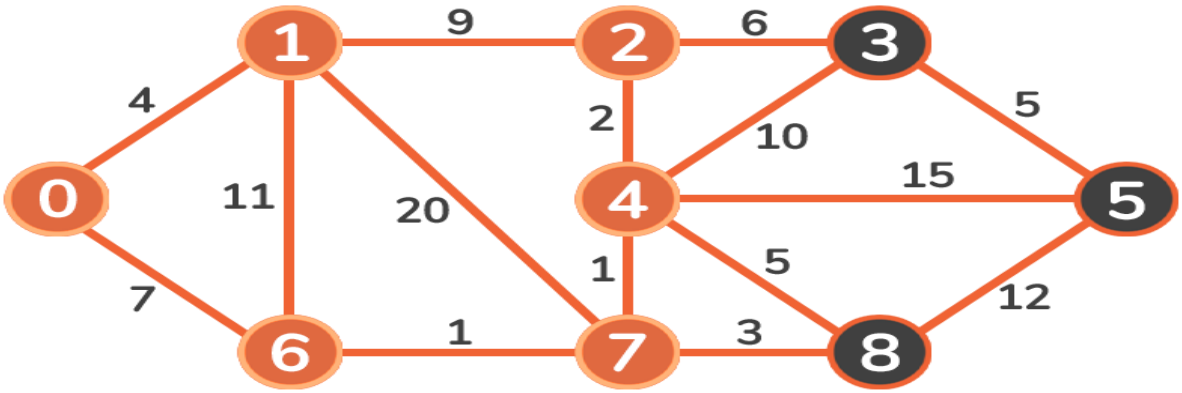
The process is continued to *Vertex 7*:

vertex	cost to get to it from vertex 0
0	**0**
1	**4**
2	13
3	inf
4	9
5	inf
6	**7**
7	**8**
8	11

And again, to *Vertex 4*:

vertex	cost to get to it from vertex 0
0	**0**
1	**4**
2	11
3	19
4	**9**
5	24
6	**7**
7	**8**
8	11

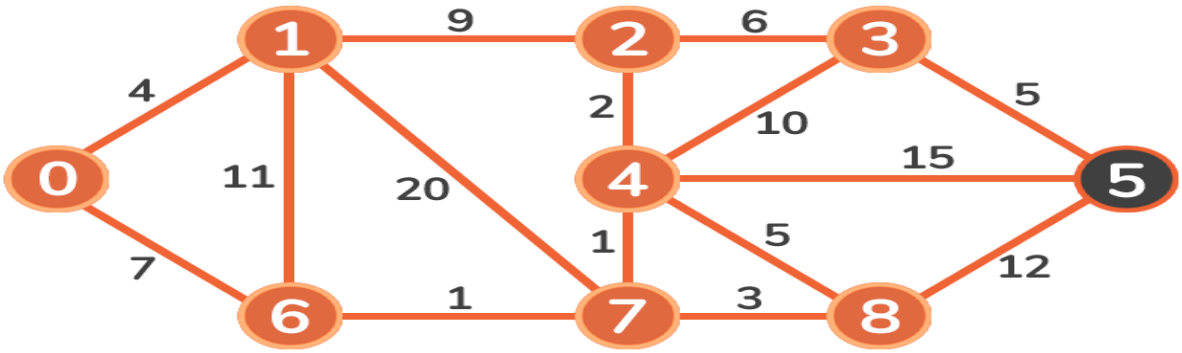
And again, to *Vertex 2*:



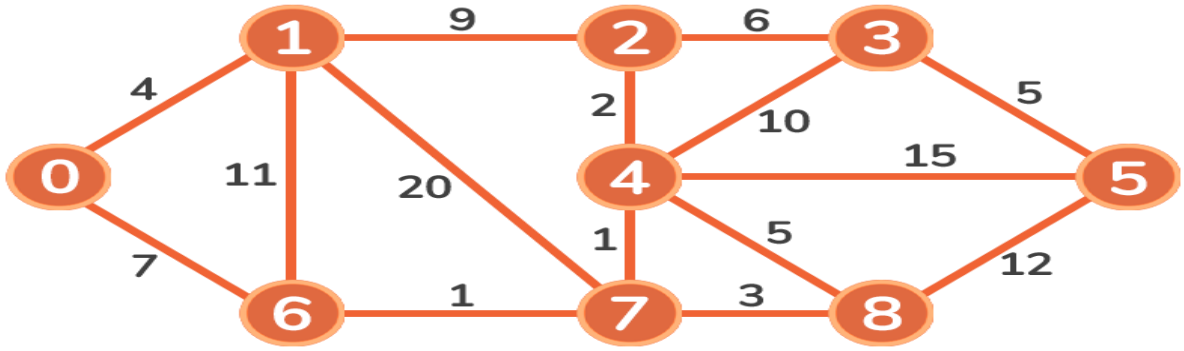
The only vertex we're going to consider is *Vertex 3*. Since 11 + 6 < 19, the cost of vertex 3 is updated.
Then, we proceed to *Vertex 8*:
Finally, we're updating the *Vertex 5*:

vertex	cost to get to it from vertex 0
0	**0**
1	**4**
2	**11**
3	17
4	**9**
5	24
6	**7**
7	**8**
8	**11**

We've updated the vertices in the loop-like structure in the end - so now we just have to traverse it - first to *Vertex 3*:



And finally, to the *Vertex 5*:



There are no more unvisited vertices that may need an update. Our final costs represents the shortest paths from node 0 to every other node in the graph:

vertex	cost to get to it from vertex 0
0	0
1	4
2	11
3	17
4	9
5	24
6	7
7	8
8	11

Program :

```
import sys

# Class to represent a graph

class Graph:

    def __init__(self, vertices):

        self.V = vertices

        self.graph = [[0] * vertices for _ in range(vertices)]


    # Function to find the vertex with the minimum distance value,

    # from the set of vertices not yet included in the shortest path tree

    def min_distance(self, dist, shortest_path_set):

        min_dist = sys.maxsize

        min_dist_index = -1
```

```
    for v in range(self.V):
        if dist[v] < min_dist and not shortest_path_set[v]:
            min_dist = dist[v]
            min_dist_index = v

    return min_dist_index

# Function to print the final shortest path tree
def print_solution(self, dist):
    print("Vertex \t\t Distance from Source")
    for v in range(self.V):
        print(v, "\t\t", dist[v])

# Function that implements Dijkstra's algorithm for a given graph and source vertex
def dijkstra(self, source):
    dist = [sys.maxsize] * self.V  # Initialize all distances as infinite
    dist[source] = 0  # Distance from source vertex to itself is always 0
    shortest_path_set = [False] * self.V

    for _ in range(self.V - 1):
        # Find the vertex with the minimum distance value
        u = self.min_distance(dist, shortest_path_set)
        shortest_path_set[u] = True

        # Update dist[v] if there is a shorter path from source to v through u
        for v in range(self.V):
            if (
                self.graph[u][v] > 0  # There is an edge from u to v
                and not shortest_path_set[v]  # v is not included in the shortest path tree
                and dist[u] != sys.maxsize  # Current distance of u is not infinity
                and dist[u] + self.graph[u][v] < dist[v]  # New path is shorter than the current distance
```

```
                ):
                    dist[v] = dist[u] + self.graph[u][v]

        self.print_solution(dist)

# Test the implementation
if __name__ == "__main__":
    # Create a graph
    graph = Graph(9)
    graph.graph = [
        [0, 4, 0, 0, 0, 0, 0, 8, 0],
        [4, 0, 8, 0, 0, 0, 0, 11, 0],
        [0, 8, 0, 7, 0, 4, 0, 0, 2],
        [0, 0, 7, 0, 9, 14, 0, 0, 0],
        [0, 0, 0, 9, 0, 10, 0, 0, 0],
        [0, 0, 4, 14, 10, 0, 2, 0, 0],
        [0, 0, 0, 0, 0, 2, 0, 1, 6],
        [8, 11, 0, 0, 0, 0, 1, 0, 7],
        [0, 0, 2, 0, 0, 0, 6, 7, 0],
    ]
    source_vertex = 0
    graph.dijkstra(source_vertex)
```

OUTPUT :

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Lab : 6

Aim : Find a mother vertex in any given graph

A mother vertex in a graph G = (V, E) is a vertex v such that all other vertices in G can be reached by a path from v.

We can consider:

- ***Case 1:- Undirected Connected Graph:*** In this case, all the vertices are mother vertices as we can reach to all the other nodes in the graph.
- ***Case 2:- Undirected/Directed Disconnected Graph***: In this case, there is no mother vertices as we cannot reach to all the other nodes in the graph.
- ***Case 3:- Directed Connected Graph***: In this case, we have to find a vertex -v in the graph such that we can reach to all the other nodes in the graph through a directed path.

We can find a mother vertex in O(V+E) time. In a graph of strongly connected components, mother vertices are always vertices of source component in component graph. The idea is based on below fact.

If there exist mother vertex (or vertices), then one of the mother vertices is the last finished vertex in DFS. (Or a mother vertex has the maximum finish time in DFS traversal).
A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

Algorithm=

1. Do DFS traversal of the given graph. While doing traversal keep track of last finished vertex 'v'. This step takes O(V+E) time.
2. If there exist mother vertex (or vetices), then v must be one (or one of them). Check if v is a mother vertex by doing DFS/BFS from v. This step also takes O(V+E) time.

Below is implementation of above algorithm.

Program :

```
from collections import defaultdict


class Graph:

  def __init__(self, vertices):

    self.vertices = vertices

    self.adj_list = defaultdict(list)


  def add_edge(self, u, v):

    self.adj_list[u].append(v)
```

```
    def DFS(self, v, visited):
        visited[v] = True

        for neighbor in self.adj_list[v]:
            if not visited[neighbor]:
                self.DFS(neighbor, visited)

    def find_mother_vertex(self):
        visited = [False] * self.vertices
        last_v = 0

        for v in range(self.vertices):
            if not visited[v]:
                self.DFS(v, visited)
                last_v = v

        visited = [False] * self.vertices
        self.DFS(last_v, visited)

        if any(not visited[v] for v in range(self.vertices)):
            return -1  # No mother vertex found

        return last_v

# Example usage
graph = Graph(7)
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 3)
graph.add_edge(4, 1)
graph.add_edge(6, 4)
```

```
graph.add_edge(5, 6)
graph.add_edge(5, 2)
graph.add_edge(6, 0)

mother_vertex = graph.find_mother_vertex()

if mother_vertex != -1:
    print("A mother vertex is:", mother_vertex)
else:
    print("No mother vertex found in the graph.")
```

OUTPUT :

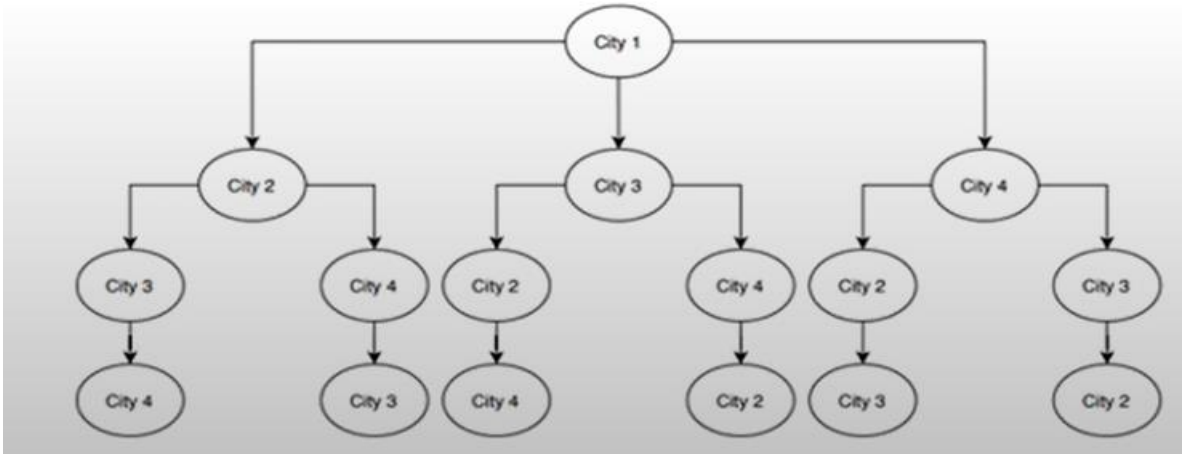
A mother vertex is: 5

Lab : 7

Aim : Solve the Travelling Salesman Problem using Genetic Algorithm in Python.

	Nashik	Mumbai	Pune	Surat
Nashik	0	166	210	235
Mumbai		0	147	278
Pune			0	418
Surat				0

No. of Possible path= 4!=24



Distance between 2 cities A□B

Sqrt(X^2 + Y^2)

=Sqrt($(X1-X2)^2$ + $(Y1-Y2)^2$)

What are heuristics algorithms?

While solving large scale linear/integer problems, it becomes extensively difficult to solve or even reach a feasible solution within the prescribed practical duration. To mitigate such issues, it is a common practice in optimization community to resort various heuristics algorithms and reach a feasible solution which may or may not be an optimum solution. Some of the heuristic algorithms are listed below:

- Greedy Search
- Tabu Search
- Breadth first Search
- Depth first Search
- Genetic Algorithm
- Particle Swarm Optimization
- Bee Colony Optimization

Heuristics algorithms are meant to find an approximate solution as the search algorithm does not traverse through all the possible solution.

Genetic Algorithm-

GA is a search-based algorithm inspired by Charles Darwin's theory of natural evolution. GA follows the notion of natural selection. The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

Genetic algorithms are heuristic search algorithms inspired by the process that supports the evolution of life. The algorithm is designed to replicate the natural selection process to carry generation, i.e. survival of the fittest of beings. Standard genetic algorithms are divided into five phases which are:

1. Creating initial population.
2. Calculating fitness.
3. Selecting the best genes.
4. Crossing over.
5. Mutating to introduce variations.

These algorithms can be implemented to find a solution to the optimization problems of various types. One such problem is the Traveling Salesman Problem. The problem says that a salesman is given a set of cities, he has to find the shortest route to as to visit each city exactly once and return to the starting city.

In the implementation, cities are taken as genes, string generated using these characters is called a chromosome, while a fitness score which is equal to the path length of all the cities mentioned, is used to target a population.

Fitness Score is defined as the length of the path described by the gene. Lesser the path length fitter is the gene. The fittest of all the genes in the gene pool survive the population test and move to the next iteration. The number of iterations depends upon the value of a cooling variable. The value of the cooling variable keeps on decreasing with each iteration and reaches a threshold after a certain number of iterations.

Algorithm:

1. Initialize the population randomly.

2. Determine the fitness of the chromosome.

3. Until done repeat:

    1. Select parents.

    2. Perform crossover and mutation.

    3. Calculate the fitness of the new population.

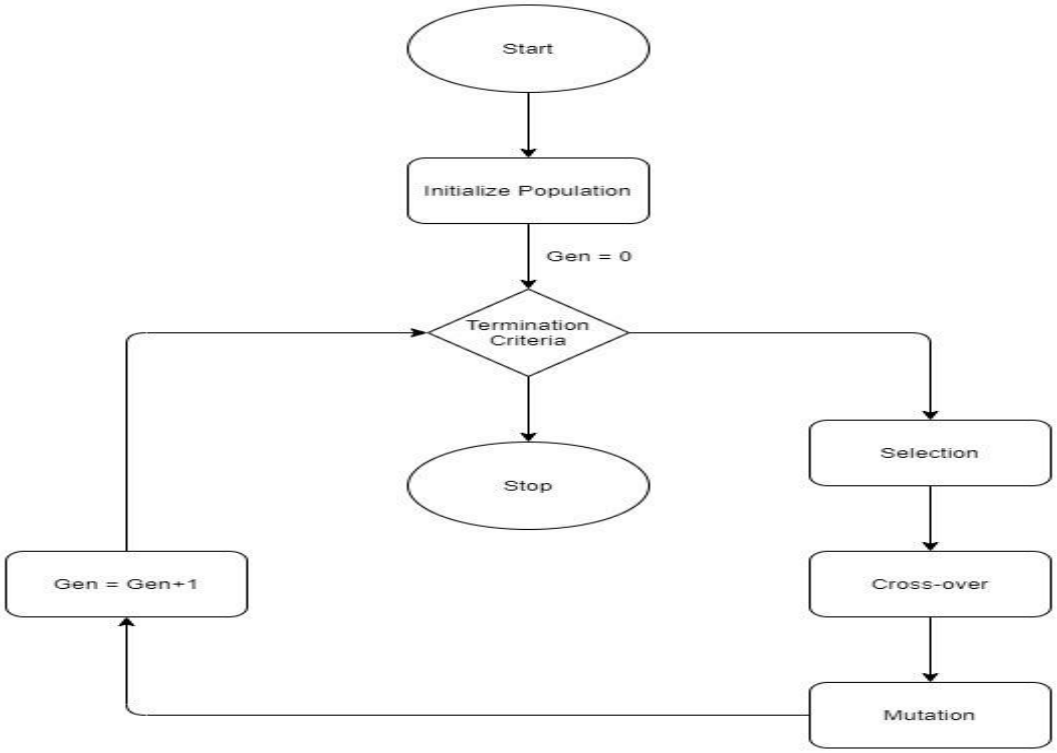
    4. Append it to the gene pool.

We can use this notion for search-based algorithm. We can formally state this process in as following phases:

1. Initial population
2. Fitness Function

3. Selection
4. Cross-over
5. Mutation

The basic flow of GA can be represented by this diagram:



Program :

```
import random

import numpy as np


# Define the coordinates of the cities

cities = np.array([[0, 0], [1, 2], [3, 4], [6, 1], [7, 3]])


# Define the parameters

population_size = 50

mutation_rate = 0.01

num_generations = 100


# Create an initial population

def create_population(size):

    population = []
```

```
    for _ in range(size):
        individual = np.arange(len(cities))
        np.random.shuffle(individual)
        population.append(individual)
    return population

# Calculate the total distance of a route
def calculate_distance(route):
    total_distance = 0
    for i in range(len(route) - 1):
        city_a = cities[route[i]]
        city_b = cities[route[i + 1]]
        distance = np.linalg.norm(city_a - city_b)
        total_distance += distance
    return total_distance

# Select parents for crossover
def select_parents(population, num_parents):
    fitness_scores = []
    for individual in population:
        fitness_scores.append(1 / calculate_distance(individual))
    selected_indices = np.random.choice(len(population), num_parents, replace=False,
p=fitness_scores/sum(fitness_scores))
    parents = [population[i] for i in selected_indices]
    return parents

# Perform crossover to create offspring
def crossover(parents):
    crossover_point = len(cities) // 2
    parent_a, parent_b = parents
```

```
    child = np.concatenate((parent_a[:crossover_point], np.setdiff1d(parent_b,
parent_a[:crossover_point])))
    return child

# Perform mutation on an individual
def mutate(individual):
    if random.random() < mutation_rate:
        indices = np.random.choice(len(individual), 2, replace=False)
        individual[indices[0]], individual[indices[1]] = individual[indices[1]], individual[indices[0]]
    return individual

# Create a new generation
def create_generation(population):
    new_generation = []
    num_parents = len(population) // 2
    parents = select_parents(population, num_parents)
    for _ in range(len(population)):
        offspring = crossover(random.sample(parents, 2))
        offspring = mutate(offspring)
        new_generation.append(offspring)
    return new_generation

# Run the genetic algorithm
def genetic_algorithm():
    population = create_population(population_size)
    best_distance = float('inf')
    best_route = None

    for generation in range(num_generations):
        population = create_generation(population)
        for individual in population:
```

```
            distance = calculate_distance(individual)
            if distance < best_distance:
                best_distance = distance
                best_route = individual

    print("Best route:", best_route)
    print("Best distance:", best_distance)
genetic_algorithm()
```

OUTPUT :

Best route: [0 1 2 3 4]
Best distance: 11.543203766865055

Lab : 8

Aim : Which algorithm would you use to find shortest path from one city to another

Which algorithm would you use to find the shortest path from one city to another?

- The current problem at hand is to find the shortest path from one city to another.
- The constructs of this problem would include ,cities which can be stated as nodes or vertices, and the distance between the cities which can be stated as edges.
- Each edge would contain a certain value which would denote the distance.
    - In this case ,since the value is distance there is no need for negative weights.
    - Since we aren't aware of the cities ,lets assume that the paths between the two cities are many.
    - We'll also need the time complexity to be as less as possible.

Taking the above points into consideration the optimal choice would be Dijkstra's algorithm.as it satisfies all the above conditions.

Program :

```
import sys

import heapq

# Define the graph (distances between cities)

graph = {

   'A': {'B': 1, 'C': 3, 'D': 4},

   'B': {'A': 1, 'C': 2, 'D': 3},

   'C': {'A': 3, 'B': 2, 'D': 5},

   'D': {'A': 4, 'B': 3, 'C': 5}

}


# Function to find the shortest path using Dijkstra's algorithm

def dijkstra(graph, start, end):

   distances = {city: sys.maxsize for city in graph}

   distances[start] = 0

   queue = [(0, start)]

   heapq.heapify(queue)

   visited = set()


   while queue:
```

```
        current_distance, current_city = heapq.heappop(queue)

        if current_city == end:
            break

        if current_distance > distances[current_city]:
            continue

        visited.add(current_city)

        for neighbor, weight in graph[current_city].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    if distances[end] == sys.maxsize:
        return None

    path = []
    city = end
    while city != start:
        path.append(city)
        city = min(graph[city], key=lambda x: distances[x])

    path.append(start)
    path.reverse()

    return path

# Run the program
```

```
start_city = 'A'

end_city = 'D'

shortest_path = dijkstra(graph, start_city, end_city)


if shortest_path is None:

    print(f"No path found between {start_city} and {end_city}")

else:

    print(f"Shortest path between {start_city} and {end_city}: {' -> '.join(shortest_path)}")

    print(f"Shortest distance: {sum(graph[shortest_path[i]][shortest_path[i+1]] for i in
range(len(shortest_path)-1))}")
```

OUTPUT :

```
Shortest path between A and D: A -> D
Shortest distance: 4
```

Lab : 9

Aim : Implement Dijkstra’s algorithm using Python.

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works
Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.
Djikstra's algorithm pseudocode
We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.
We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.
A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Program :

class Graph():

```
def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)]
                  for row in range(vertices)]

def printSolution(self, dist):
    print("Vertex \t Distance from Source")
    for node in range(self.V):
        print(node, "\t\t", dist[node])

# A utility function to find the vertex with
# minimum distance value, from the set of vertices
# not yet included in shortest path tree
def minDistance(self, dist, sptSet):

    # Initialize minimum distance for next node
    min = 1e7

    # Search not nearest vertex not in the
    # shortest path tree
    for v in range(self.V):
        if dist[v] < min and sptSet[v] == False:
            min = dist[v]
            min_index = v

    return min_index

# Function that implements Dijkstra's single source
# shortest path algorithm for a graph represented
# using adjacency matrix representation
```

```
def dijkstra(self, src):

        dist = [1e7] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shortest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shortest path tree
            for v in range(self.V):
                if (self.graph[u][v] > 0 and
                    sptSet[v] == False and
                    dist[v] > dist[u] + self.graph[u][v]):
                     dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

# Driver program
g = Graph(9)
```

```
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
        [4, 0, 8, 0, 0, 0, 0, 11, 0],
        [0, 8, 0, 7, 0, 4, 0, 0, 2],
        [0, 0, 7, 0, 9, 14, 0, 0, 0],
        [0, 0, 0, 9, 0, 10, 0, 0, 0],
        [0, 0, 4, 14, 10, 0, 2, 0, 0],
        [0, 0, 0, 0, 0, 2, 0, 1, 6],
        [8, 11, 0, 0, 0, 0, 1, 0, 7],
        [0, 0, 2, 0, 0, 0, 6, 7, 0]
        ]

graph.dijkstra(0)
```

OUTPUT :

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Lab : 10

Aim : Stochastic Hill Climbing in Python

Stochastic Hill climbing is an optimization algorithm.

It makes use of randomness as part of the search process. This makes the algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well.

It is also a local search algorithm, meaning that it modifies a single solution and searches the relatively local area of the search space until the local optima is located. This means that it is appropriate on unimodal optimization problems or for use after the application of a global optimization algorithm.

Hill Climbing Algorithm

The stochastic hill climbing algorithm is a stochastic local search optimization algorithm.

It takes an initial point as input and a step size, where the step size is a distance within the search space.

The algorithm takes the initial point as the current best candidate solution and generates a new point within the step size distance of the provided point. The generated point is evaluated, and if it is equal or better than the current point, it is taken as the current point.

The generation of the new point uses randomness, often referred to as Stochastic Hill Climbing. This means that the algorithm can skip over bumpy, noisy, discontinuous, or deceptive regions of the response surface as part of the search.

Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.

It is important that different points with equal evaluation are accepted as it allows the algorithm to continue to explore the search space, such as across flat regions of the response surface. It may also be helpful to put a limit on these so-called "*sideways*" moves to avoid an infinite loop.

This process continues until a stop condition is met, such as a maximum number of function evaluations or no improvement within a given number of function evaluations.

The algorithm takes its name from the fact that it will (stochastically) climb the hill of the response surface to the local optima. This does not mean it can only be used for maximizing objective functions; it is just a name. In fact, typically, we minimize functions instead of maximize them.

The hill-climbing search algorithm (steepest-ascent version) […] is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbour has a higher value.

Hill Climbing Algorithm Implementation

At the time of writing, the SciPy library does not provide an implementation of stochastic hill climbing.

Nevertheless, we can implement it ourselves.

First, we must define our objective function and the bounds on each input variable to the objective function. The objective function is just a Python function we will name *objective()*. The bounds will be a

2D array with one dimension for each input variable that defines the minimum and maximum for the variable.

For example, a one-dimensional objective function and bounds would be defined as follows:

Program :

```
from numpy import asarray
from numpy.random import rand
from matplotlib import pyplot
# objective function
def objective(x):
	return 0

# define range for input
bounds = asarray([[-5.0, 5.0]])

solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
# evaluate the initial point
solution_eval = objective(solution)

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
	# generate an initial point
	solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
	# evaluate the initial point
	solution_eval = objective(solution)
	# run the hill climb
	for i in range(n_iterations):
		# take a step
		candidate = solution + randn(len(bounds)) * step_size
		# evaluate candidate point
		candidte_eval = objective(candidate)
```

```
            # check if we should keep the new point
            if candidte_eval <= solution_eval:
                # store the new point
                solution, solution_eval = candidate, candidte_eval
                # report progress
                print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]


# convex unimodal optimization function
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
    return x[0]**2.0

# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define optimal input value
x_optima = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=x_optima, ls='--', color='red')
# show the plot
pyplot.show()
```

OUTPUT :