**Ishita Hardasmalani**

**FINAL CODES FOR PRACTICAL**

1. **DFS FOR GRAPH**

def dfs(graph, start, goal, visited=None, path=None, open\_list=None, closed\_list=None):

if visited is None:

visited = set()

if path is None:

path = []

if open\_list is None:

open\_list = []

if closed\_list is None:

closed\_list = []

visited.add(start)

open\_list.append(start)

path = path + [start]

print("Iteration", len(open\_list) + len(closed\_list))

print("Current Node:", start)

print("Open List:", open\_list)

print("Closed List:", closed\_list)

if start == goal:

return path

for neighbor in graph[start]:

if neighbor not in visited:

new\_path = dfs(graph, neighbor, goal, visited, path, open\_list, closed\_list)

if new\_path:

return new\_path

open\_list.remove(start)

closed\_list.append(start)

return None

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

print("DFS Path:")

if start\_node not in graph or goal\_node not in graph:

print("Start node or goal node not found in the graph.")

else:

open\_list = []

closed\_list = []

path = dfs(graph, start\_node, goal\_node, open\_list=open\_list, closed\_list=closed\_list)

if path:

print("Path from", start\_node, "to", goal\_node, ":", ' -> '.join(path))

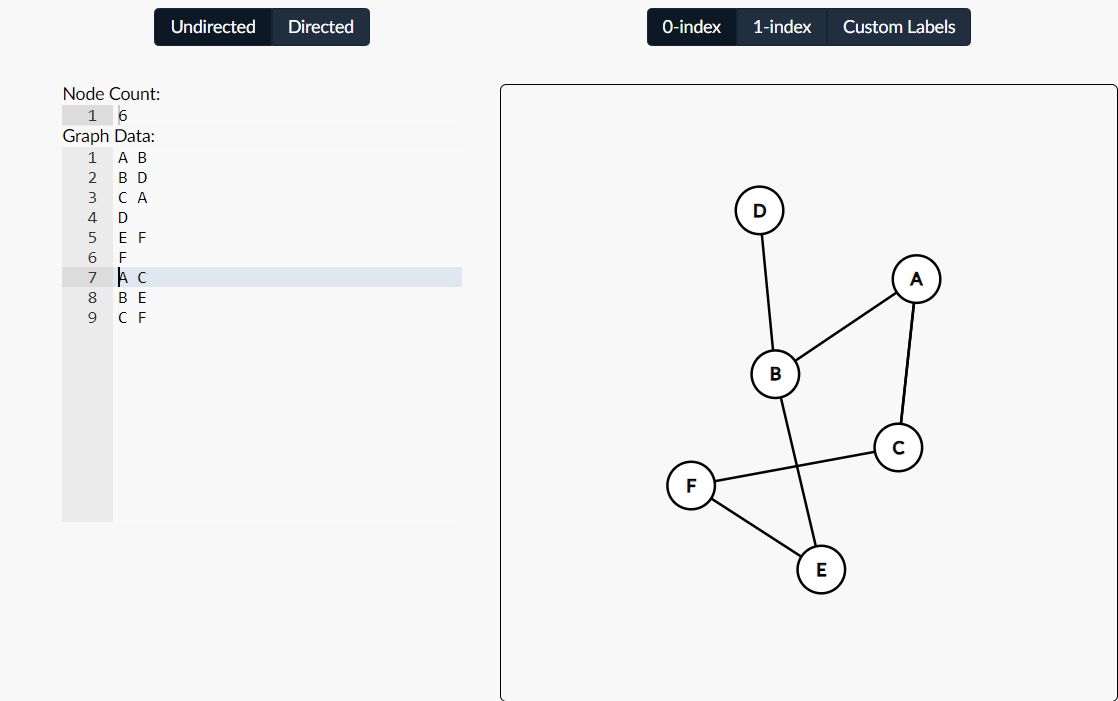
else:

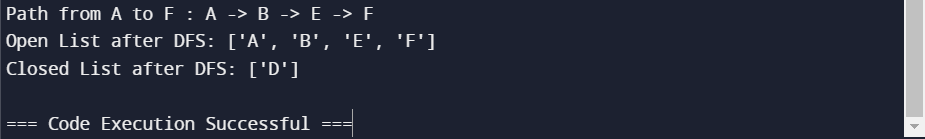
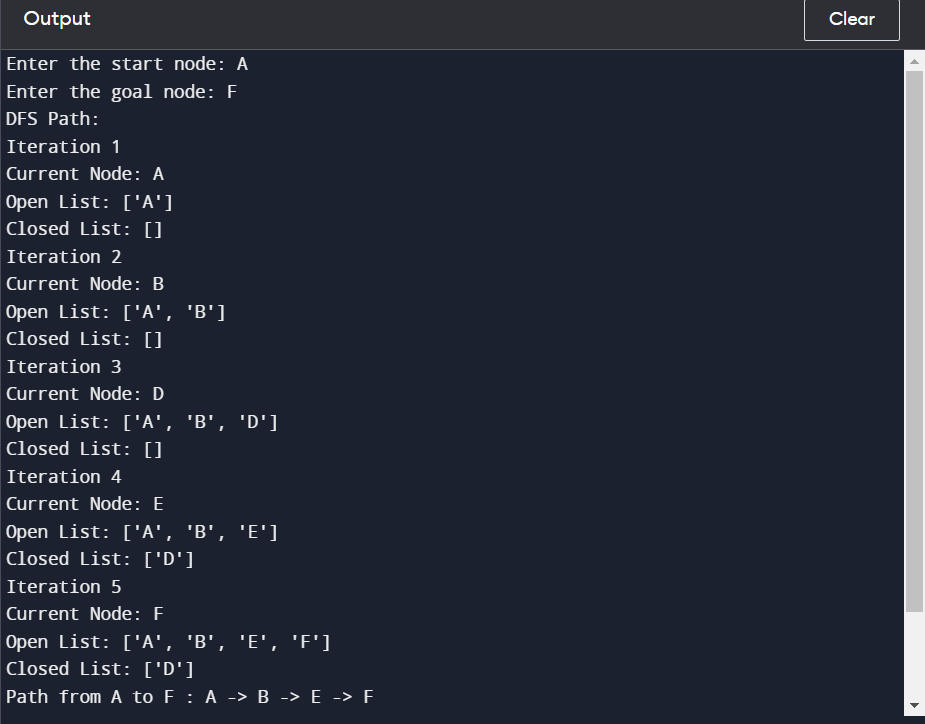
print("Path from", start\_node, "to", goal\_node, "not found.")

print("Open List after DFS:", open\_list)

print("Closed List after DFS:", closed\_list)

GRAPH:





1. **DFS FOR TREE**

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def dfs\_tree(node, goal, path=None, open\_list=None, closed\_list=None):

if path is None:

path = []

if open\_list is None:

open\_list = []

if closed\_list is None:

closed\_list = []

open\_list.append(node.value)

path.append(node.value)

print("Iteration", len(open\_list) + len(closed\_list))

print("Current Node:", node.value)

print("Open List:", open\_list)

print("Closed List:", closed\_list)

if node.value == goal:

return path

for child in node.children:

if child.value not in path:

new\_path = dfs\_tree(child, goal, path[:], open\_list, closed\_list)

if new\_path:

return new\_path

open\_list.remove(node.value)

closed\_list.append(node.value)

return None

root = TreeNode('A')

root.children = [TreeNode('B'), TreeNode('C'), TreeNode('D')]

root.children[0].children = [TreeNode('E'), TreeNode('F')]

root.children[1].children = [TreeNode('G')]

goal\_node = input("Enter the goal node: ").strip().upper()

print("DFS Path in Tree:")

open\_list = []

closed\_list = []

path = dfs\_tree(root, goal\_node, open\_list=open\_list, closed\_list=closed\_list)

if path:

print("Path to", goal\_node, ":", ' -> '.join(path))

else:

print("Path to", goal\_node, "not found.")

TREE:

A

/|\

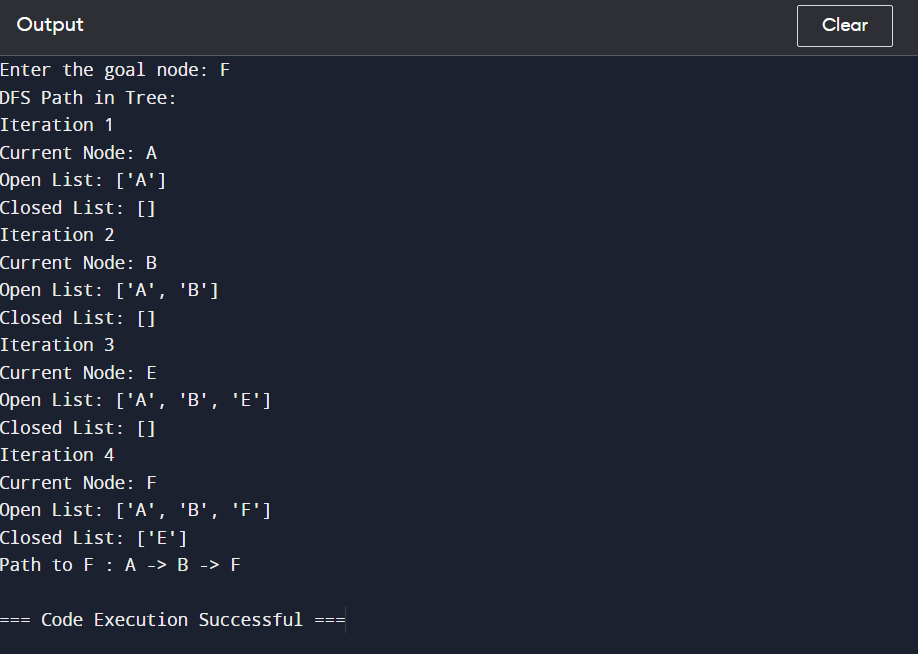
B C D

/ \

E G

/

F



1. **DLS FOR GRAPH**

def dls(graph, start, goal, max\_depth, depth=0, visited=None, path=None):

if visited is None:

visited = set()

if path is None:

path = []

visited.add(start)

path = path + [start]

if start == goal:

return path

if depth >= max\_depth:

return None

for neighbor in graph[start]:

if neighbor not in visited:

new\_path = dls(graph, neighbor, goal, max\_depth, depth + 1, visited, path)

if new\_path:

return new\_path

return None

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

max\_depth = int(input("Enter the maximum depth: ").strip())

print("DLS Path:")

if start\_node not in graph or goal\_node not in graph:

print("Start node or goal node not found in the graph.")

else:

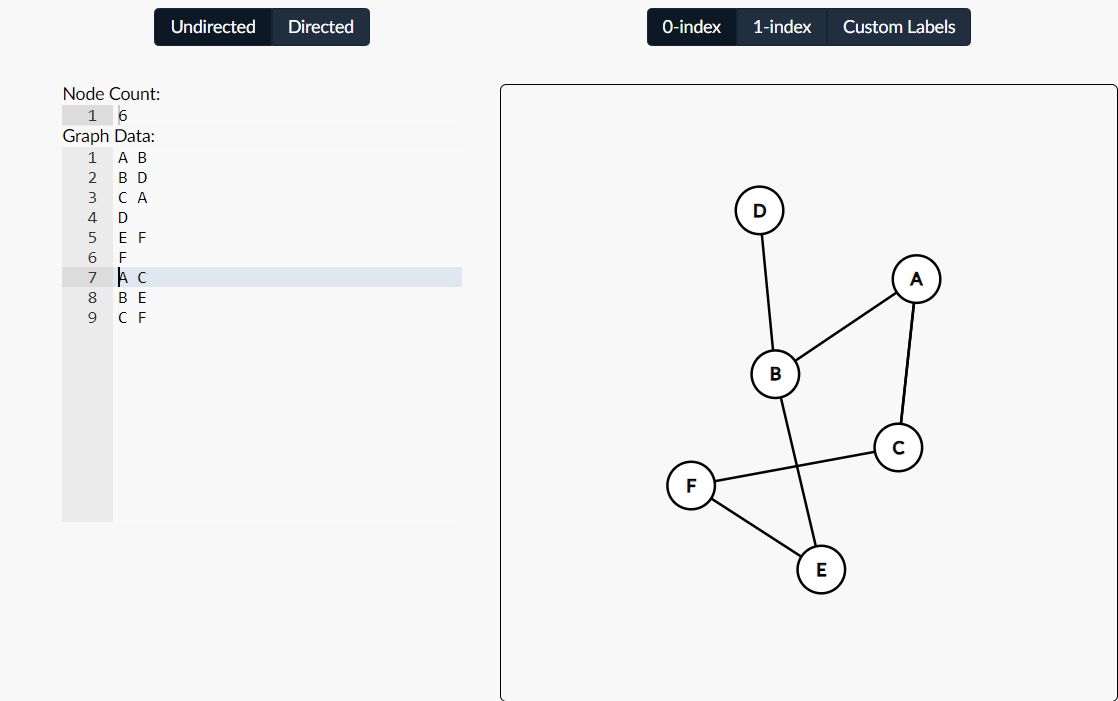
path = dls(graph, start\_node, goal\_node, max\_depth)

if path:

print("Path from", start\_node, "to", goal\_node, ":", ' -> '.join(path))

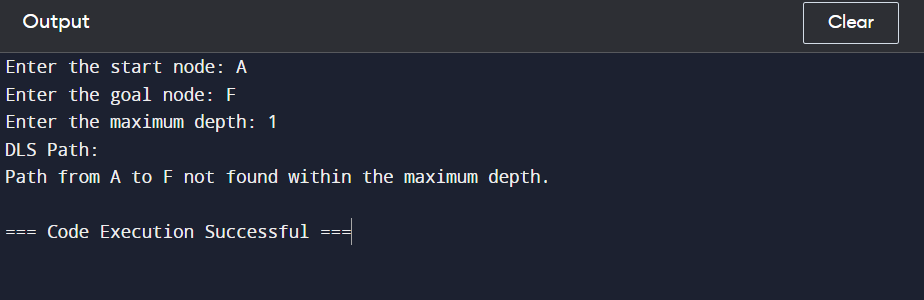
else:

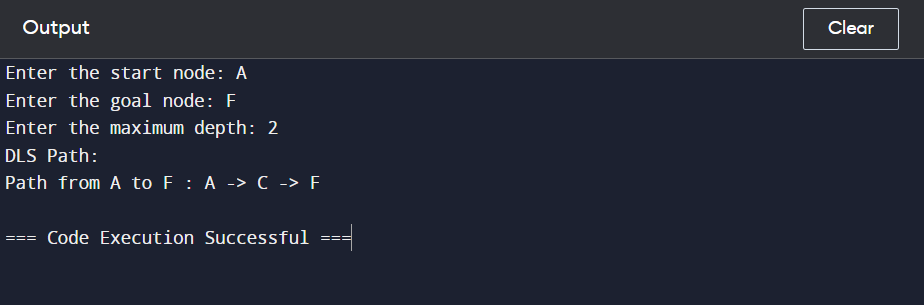
print("Path from", start\_node, "to", goal\_node, "not found within the maximum depth.")

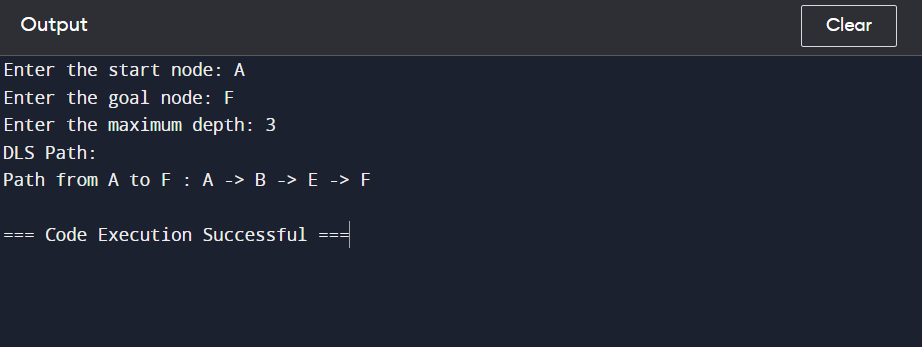
GRAPH

OUTPUT:

CASE 1



CASE 2

CASE 3

1. **DLS FOR TREE**

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def dls\_tree(node, goal, max\_depth, depth=0, path=None):

if path is None:

path = []

path.append(node.value)

if node.value == goal:

return path

if depth >= max\_depth:

return None

for child in node.children:

new\_path = dls\_tree(child, goal, max\_depth, depth + 1, path[:])

if new\_path:

return new\_path

return None

root = TreeNode('A')

root.children = [TreeNode('B'), TreeNode('C'), TreeNode('D')]

root.children[0].children = [TreeNode('E'), TreeNode('F')]

root.children[1].children = [TreeNode('G'), TreeNode('H')]

goal\_node = input("Enter the goal node: ").strip().upper()

max\_depth = int(input("Enter the maximum depth: ").strip())

print("DLS Path in Tree:")

path = dls\_tree(root, goal\_node, max\_depth)

if path:

print("Path to", goal\_node, ":", ' -> '.join(path))

else:

print("Path to", goal\_node, "not found within the maximum depth.")

TREE:

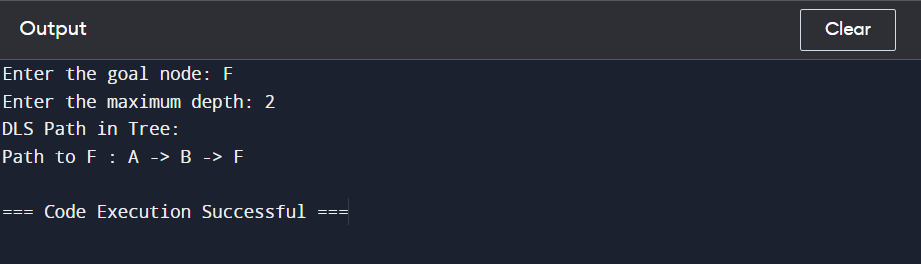
A

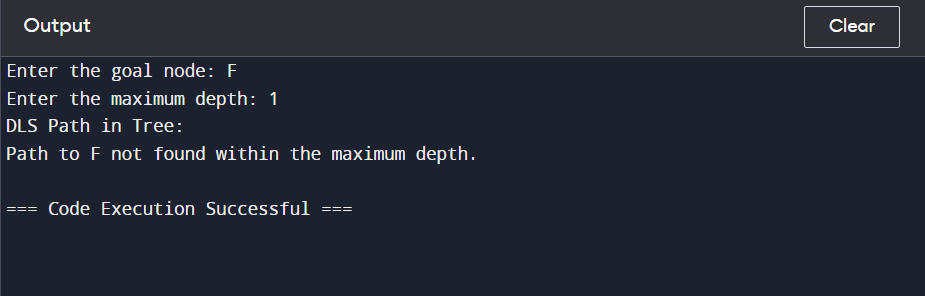


B C D



E F G H





1. **DFID FOR GRAPH**

def dfs(graph, start, goal, max\_depth, depth=0, visited=None, path=None):

if visited is None:

visited = set()

if path is None:

path = []

visited.add(start)

path = path + [start]

if start == goal:

return path

if depth >= max\_depth:

return None

for neighbor in graph[start]:

if neighbor not in visited:

new\_path = dfs(graph, neighbor, goal, max\_depth, depth + 1, visited, path)

if new\_path:

return new\_path

return None

def dfid(graph, start, goal):

depth = 0

while True:

result = dfs(graph, start, goal, depth)

if result is not None:

return result

depth += 1

# Example graph represented as an adjacency list

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

print("DFID Path:")

if start\_node not in graph or goal\_node not in graph:

print("Start node or goal node not found in the graph.")

else:

path = dfid(graph, start\_node, goal\_node)

if path:

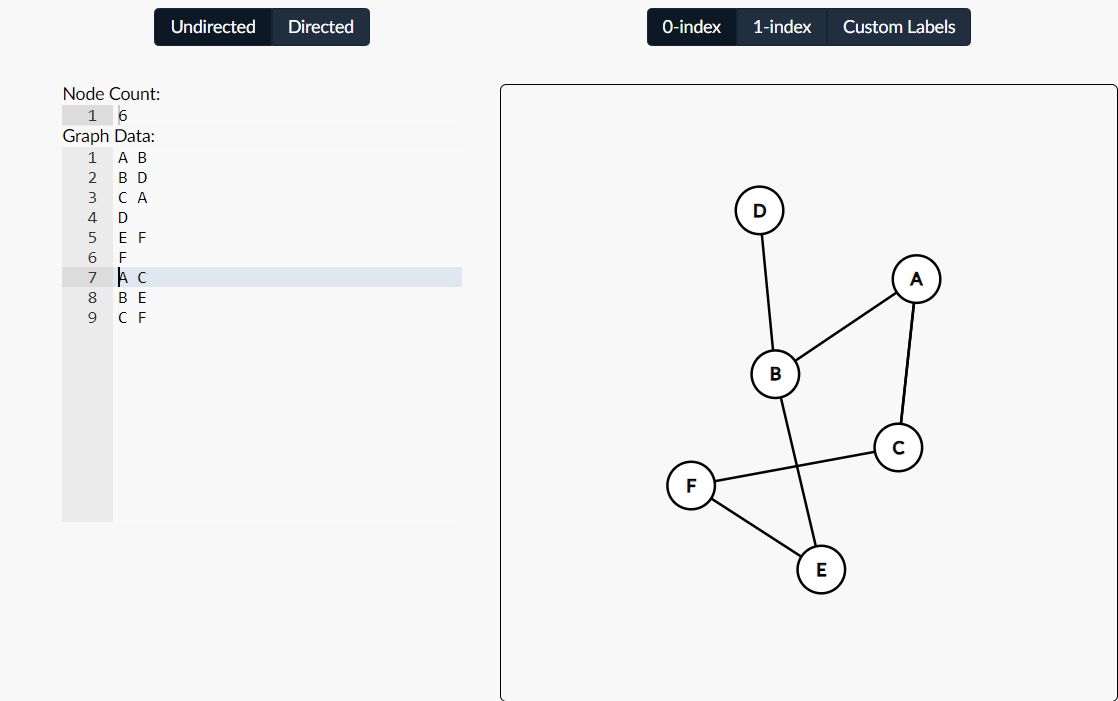
print("Path from", start\_node, "to", goal\_node, ":", ' -> '.join(path))

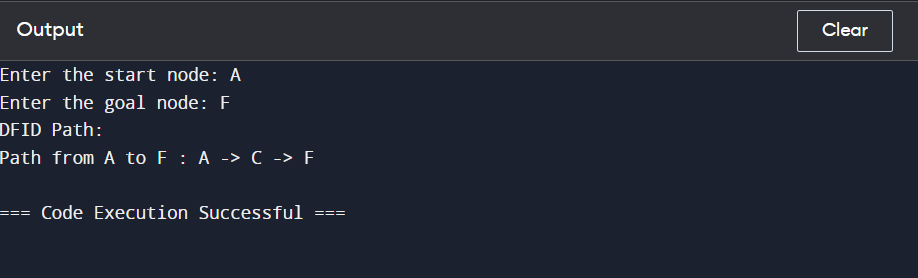
else:

print("Path from", start\_node, "to", goal\_node, "not found.")

In this code:

* **dfs** function performs Depth-First Search (DFS) with a depth limit.
* **dfid** function iterates over increasing depth limits and applies DFS until the goal is found or the maximum depth is reached.





**6.DFID FOR TREE**

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def dfs\_tree(node, goal, max\_depth, depth=0, path=None):

if path is None:

path = []

path.append(node.value)

if node.value == goal:

return path

if depth >= max\_depth:

return None

for child in node.children:

new\_path = dfs\_tree(child, goal, max\_depth, depth + 1, path[:])

if new\_path:

return new\_path

return None

def dfid\_tree(root, goal):

depth = 0

while True:

result = dfs\_tree(root, goal, depth)

if result is not None:

return result

depth += 1

# Example tree

root = TreeNode('A')

root.children = [TreeNode('B'), TreeNode('C'), TreeNode('D')]

root.children[0].children = [TreeNode('E'), TreeNode('F')]

root.children[1].children = [TreeNode('G'), TreeNode('H')]

goal\_node = input("Enter the goal node: ").strip().upper()

print("DFID Path in Tree:")

path = dfid\_tree(root, goal\_node)

if path:

print("Path to", goal\_node, ":", ' -> '.join(path))

else:

print("Path to", goal\_node, "not found.")

TREE:

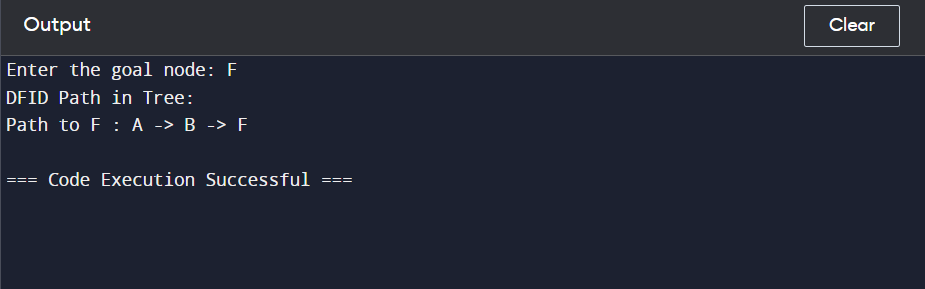
A



B C D



E F G H



1. **Breadth First Search for Graph**

from collections import deque

def bfs(graph, start, goal):

queue = deque([start])

visited = {start: None}

while queue:

node = queue.popleft()

if node == goal:

return construct\_path(visited, start, goal)

for neighbor in graph[node]:

if neighbor not in visited:

visited[neighbor] = node

queue.append(neighbor)

return None

def construct\_path(visited, start, goal):

path = []

while goal is not None:

path.append(goal)

goal = visited[goal]

return list(reversed(path))

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

print("BFS Path:")

if start\_node not in graph or goal\_node not in graph:

print("Start node or goal node not found in the graph.")

else:

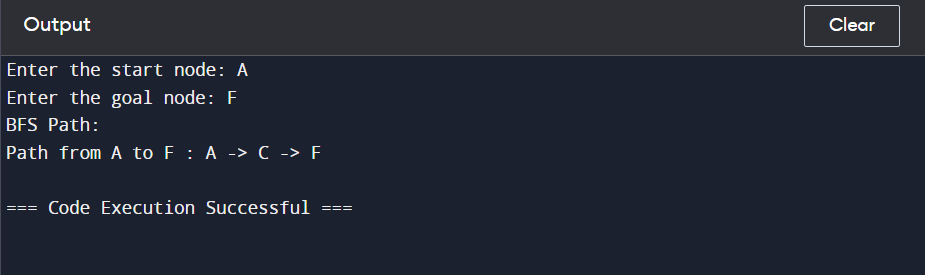
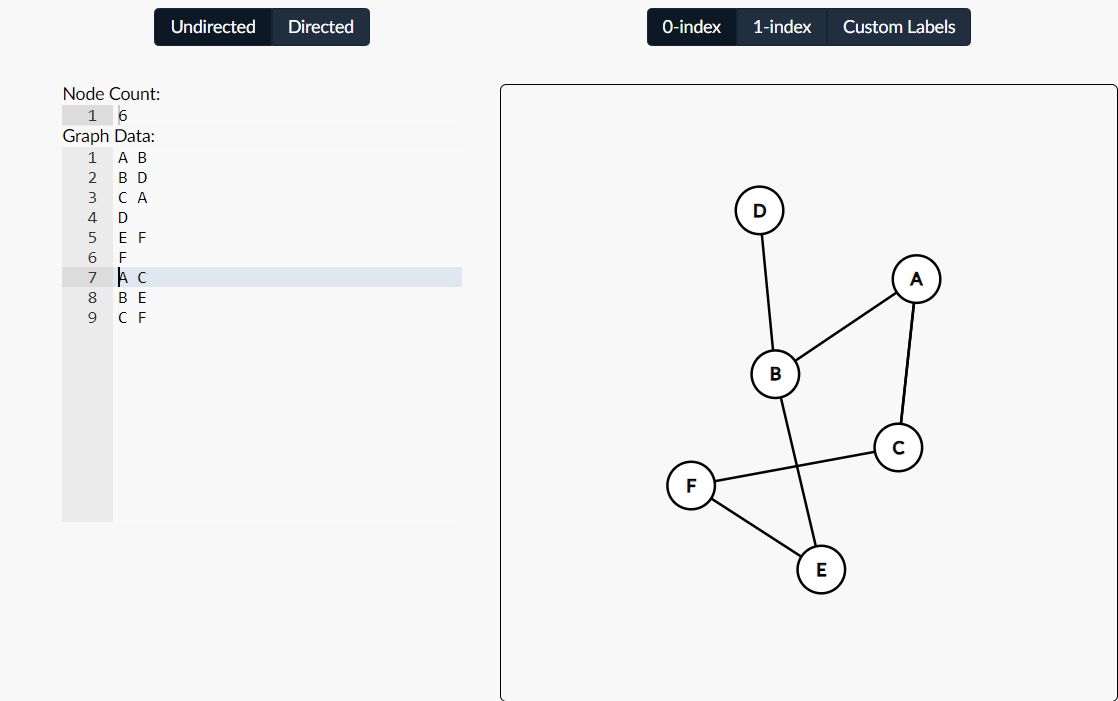
path = bfs(graph, start\_node, goal\_node)

if path:

print("Path from", start\_node, "to", goal\_node, ":", ' -> '.join(path))

else:

print("Path from", start\_node, "to", goal\_node, "not found.")



1. **Breadth First Search for Tree**

from collections import deque

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def bfs\_tree(root, goal):

queue = deque([root])

parents = {root: None}

while queue:

node = queue.popleft()

if node.value == goal:

return construct\_path(parents, node)

for child in node.children:

if child not in parents:

parents[child] = node

queue.append(child)

return None

def construct\_path(parents, goal):

path = []

while goal is not None:

path.append(goal.value)

goal = parents[goal]

return list(reversed(path))

root = TreeNode('A')

root.children = [TreeNode('B'), TreeNode('C'), TreeNode('D')]

root.children[0].children = [TreeNode('E'), TreeNode('F')]

root.children[1].children = [TreeNode('G'), TreeNode('H')]

goal\_node = input("Enter the goal node: ").strip().upper()

print("BFS Path in Tree:")

path = bfs\_tree(root, goal\_node)

if path:

print("Path to", goal\_node, ":", ' -> '.join(path))

else:

print("Path to", goal\_node, "not found.")

TREE:

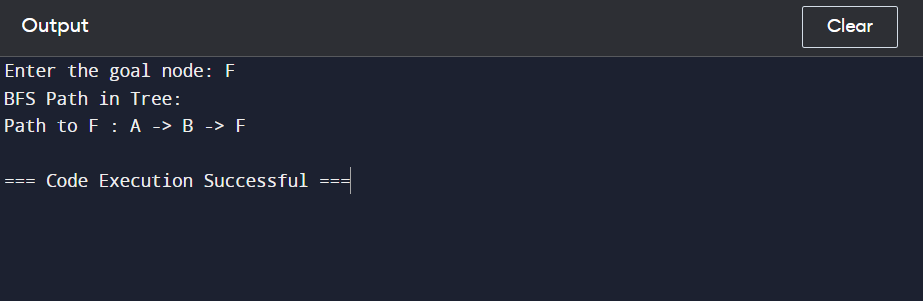
A



B C D



E F G H



1. **Greedy Best First Search Algorithm for Graph**

import heapq

def gbfs(graph, start, goal, heuristic):

visited = set()

priority\_queue = [(heuristic[start], start)]

path = {start: None}

while priority\_queue:

\_, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal:

return construct\_path(path, start, goal)

visited.add(current\_node)

for neighbor in graph[current\_node]:

if neighbor not in visited:

heapq.heappush(priority\_queue, (heuristic[neighbor], neighbor))

path[neighbor] = current\_node

return None

def construct\_path(path, start, goal):

current\_node = goal

path\_sequence = []

while current\_node:

path\_sequence.insert(0, current\_node)

current\_node = path[current\_node]

return path\_sequence if path\_sequence[0] == start else None

graph = {

'A': ['S', 'T', 'Z'],

'S': ['A', 'F', 'O', 'R'],

'T': [],

'Z': [],

'F': ['S', 'B']

}

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

heuristic = {

'A': 366,

'S': 253,

'F': 176,

'T': 329,

'O': 380,

'Z': 374,

'R': 193,

'B': 0

}

gbfs\_path = gbfs(graph, start\_node, goal\_node, heuristic)

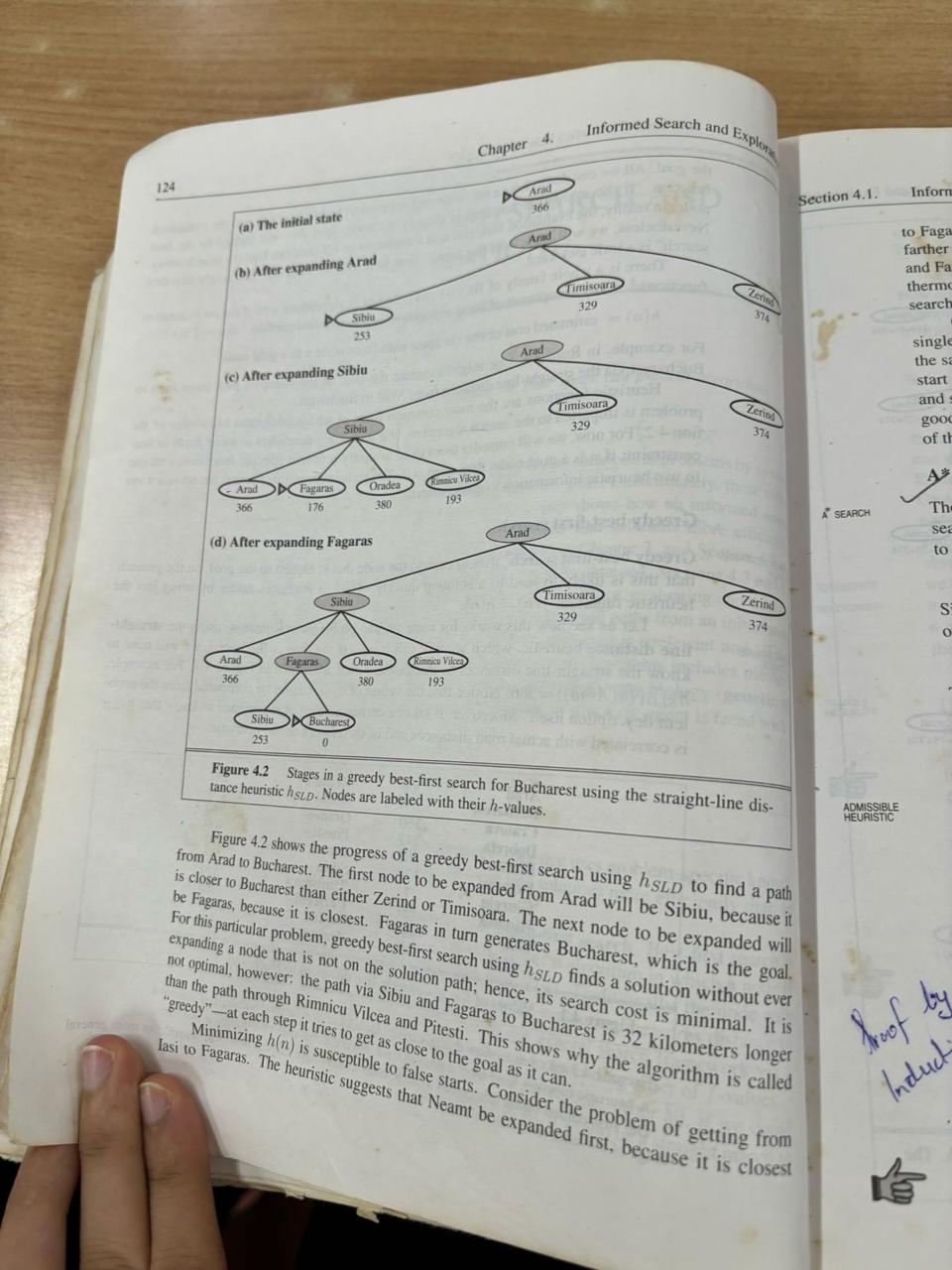
if gbfs\_path:

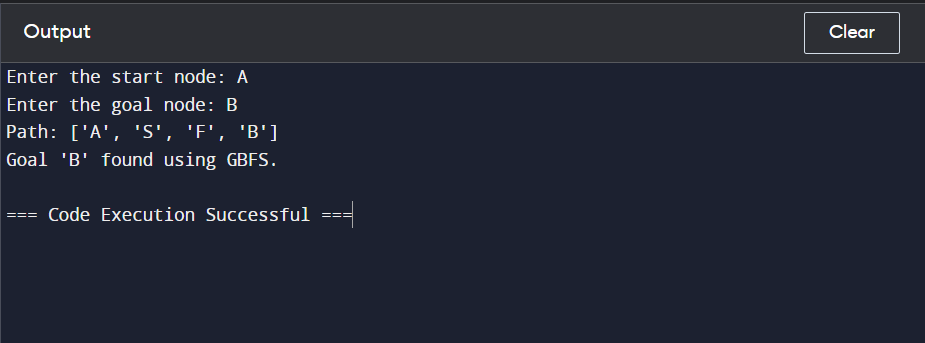
print('Path:', gbfs\_path)

print(f"Goal '{goal\_node}' found using GBFS.")

else:

print(f"Goal '{goal\_node}' not found using GBFS.")





1. **Greedy Best First Search Algorithm for Tree**

import heapq

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def gbfs\_tree(root, goal, heuristic):

visited = set()

priority\_queue = [(heuristic[root.value], root)]

path = {root.value: None}

while priority\_queue:

\_, current\_node = heapq.heappop(priority\_queue)

if current\_node.value == goal:

return construct\_path(path, root.value, goal)

visited.add(current\_node.value)

for child in current\_node.children:

if child.value not in visited:

heapq.heappush(priority\_queue, (heuristic[child.value], child))

path[child.value] = current\_node.value

return None

def construct\_path(path, start, goal):

current\_node = goal

path\_sequence = []

while current\_node:

path\_sequence.insert(0, current\_node)

current\_node = path[current\_node]

return path\_sequence if path\_sequence[0] == start else None

root = TreeNode('A')

root.children = [TreeNode('S'), TreeNode('T'), TreeNode('Z')]

root.children[0].children = [TreeNode('F'), TreeNode('O'), TreeNode('R')]

root.children[1].children = []

root.children[2].children = []

root.children[0].children[0].children = [TreeNode('B')]

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

heuristic = {

'A': 366,

'S': 253,

'F': 176,

'T': 329,

'O': 380,

'Z': 374,

'R': 193,

'B': 0

}

gbfs\_path = gbfs\_tree(root, goal\_node, heuristic)

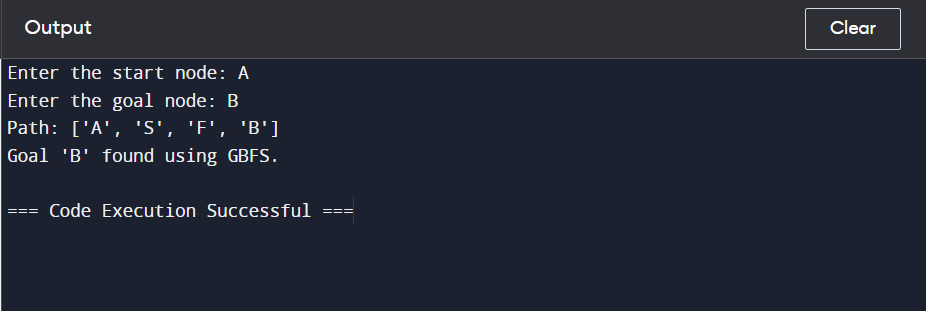
if gbfs\_path:

print('Path:', gbfs\_path)

print(f"Goal '{goal\_node}' found using GBFS.")

else:

print(f"Goal '{goal\_node}' not found using GBFS.")



1. **A\* for graph**

import heapq

def astar(graph, start, goal, heuristic, cost):

visited = set()

priority\_queue = [(heuristic[start], 0, start)] # Priority queue sorted by f-value (heuristic + cost)

path\_cost = {start: 0}

path = {start: None}

while priority\_queue:

print("OPEN LIST:", priority\_queue)

\_, current\_cost, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal:

return construct\_path(path, start, goal)

visited.add(current\_node)

for neighbor, edge\_cost in graph[current\_node].items():

total\_cost = path\_cost[current\_node] + edge\_cost

if neighbor not in visited or total\_cost < path\_cost[neighbor]:

path\_cost[neighbor] = total\_cost

heapq.heappush(priority\_queue, (total\_cost + heuristic[neighbor], total\_cost, neighbor))

path[neighbor] = current\_node

return None

def construct\_path(path, start, goal):

current\_node = goal

path\_sequence = []

while current\_node:

path\_sequence.insert(0, current\_node)

current\_node = path[current\_node]

return path\_sequence

# Example usage

graph = {

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

'B': {'D': 3, 'E': 4},

'C': {'F': 1},

'D': {},

'E': {},

'F': {}

}

start\_node = input("Enter the start node: ").upper()

goal\_node = input("Enter the goal node: ").upper()

heuristic = {

'A': 3,

'B': 2,

'C': 4,

'D': 1,

'E': 1,

'F': 0

}

cost = {

'A': 0,

'B': 1,

'C': 2,

'D': 3,

'E': 4,

'F': 5

}

astar\_path = astar(graph, start\_node, goal\_node, heuristic, cost)

print()

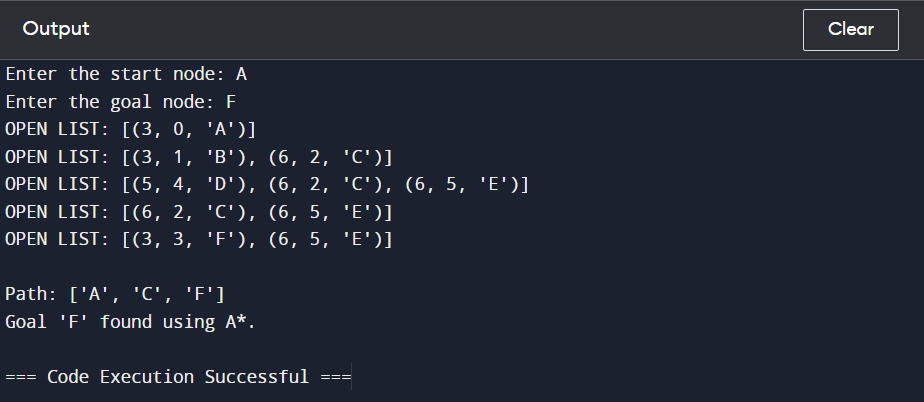
if astar\_path:

print('Path:', astar\_path)

print(f"Goal '{goal\_node}' found using A\*.")

else:

print(f"Goal '{goal\_node}' not found using A\*.")



1. **A\* for tree**

import heapq

class TreeNode:

def \_\_init\_\_(self, value, cost=None):

self.value = value

self.cost = cost

self.children = []

def \_\_repr\_\_(self):

return self.value

def astar\_tree(root, goal, heuristic):

visited = set()

priority\_queue = [(heuristic[root.value], 0, root)] # Priority queue sorted by f-value (heuristic + cost)

path\_cost = {root.value: 0}

path = {root.value: None}

while priority\_queue:

print("OPEN LIST:", [(f"{heuristic[node.value] + cost}, {cost}, {node}") for \_, cost, node in priority\_queue])

\_, current\_cost, current\_node = heapq.heappop(priority\_queue)

if current\_node.value == goal:

return construct\_path(path, root.value, goal)

visited.add(current\_node.value)

for child, edge\_cost in current\_node.children:

total\_cost = path\_cost[current\_node.value] + edge\_cost

if child.value not in visited or total\_cost < path\_cost[child.value]:

path\_cost[child.value] = total\_cost

heapq.heappush(priority\_queue, (total\_cost + heuristic[child.value], total\_cost, child))

path[child.value] = current\_node.value

return None

def construct\_path(path, start, goal):

current\_node = goal

path\_sequence = []

while current\_node:

path\_sequence.insert(0, current\_node)

current\_node = path[current\_node]

return path\_sequence

# Example usage

root = TreeNode('A')

root.children = [(TreeNode('B', 1), 1), (TreeNode('C', 2), 2)]

root.children[0][0].children = [(TreeNode('D', 3), 3), (TreeNode('E', 4), 4)]

root.children[1][0].children = [(TreeNode('F', 1), 1)]

start\_node = input("Enter the start node: ").strip().upper()

goal\_node = input("Enter the goal node: ").strip().upper()

heuristic = {

'A': 3,

'B': 2,

'C': 4,

'D': 1,

'E': 1,

'F': 0

}

astar\_path = astar\_tree(root, goal\_node, heuristic)

print()

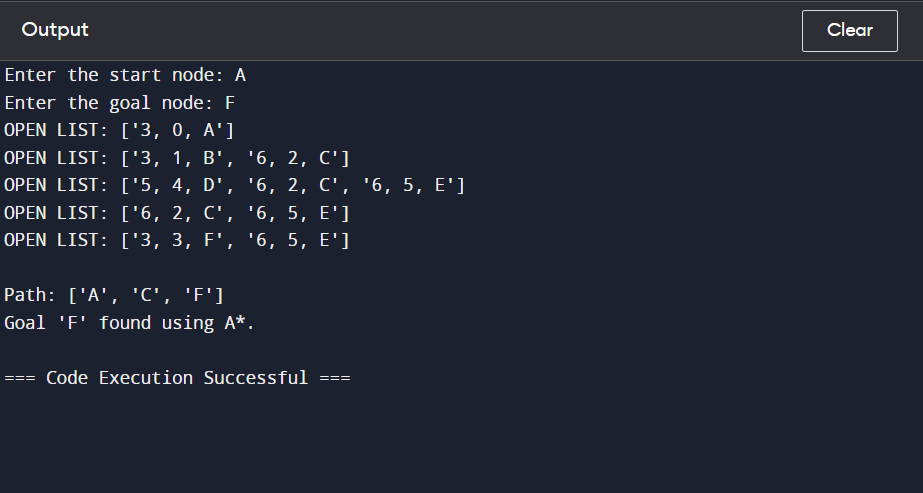
if astar\_path:

print('Path:', astar\_path)

print(f"Goal '{goal\_node}' found using A\*.")

else:

print(f"Goal '{goal\_node}' not found using A\*.")



1. **Hill Climbing Algorithm**

**f(x) = -x\*\*2**

import random

# Define the objective function

def objective\_function(x):

return -x \*\* 2

# Generate initial solution

def generate\_initial\_solution():

return random.randint(-100, 100)

# Generate neighbour solutions

def generate\_neighbours(solution):

neighbours = []

for delta in [-1, 1]:

neighbours.append(solution + delta)

return neighbours

# Find the best neighbour

def get\_best\_neighbour(neighbours):

best\_neighbour = neighbours[0]

best\_quality = objective\_function(best\_neighbour)

for neighbour in neighbours[1:]:

neighbour\_quality = objective\_function(neighbour)

if neighbour\_quality > best\_quality:

best\_quality = neighbour\_quality

best\_neighbour = neighbour

return best\_neighbour

# Hill climbing algorithm

def hill\_climbing():

current\_solution = generate\_initial\_solution()

while True:

neighbours = generate\_neighbours(current\_solution)

best\_neighbour = get\_best\_neighbour(neighbours)

if objective\_function(best\_neighbour) <= objective\_function(current\_solution):

return current\_solution

current\_solution = best\_neighbour

# Main function

def main():

best\_solution = hill\_climbing()

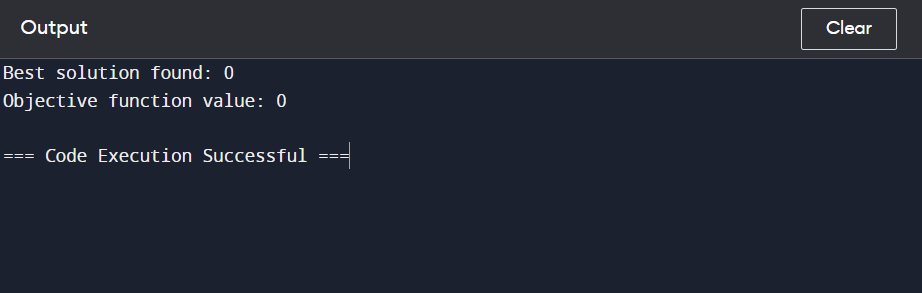
print("Best solution found:", best\_solution)

print("Objective function value:", objective\_function(best\_solution))

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

main()



**f(x) = sinx**

import random

import math

# Define the objective function

def objective\_function(x):

return math.sin(x)

# Generate initial solution

def generate\_initial\_solution():

return random.uniform(-math.pi, math.pi)

# Generate neighbour solutions

def generate\_neighbours(solution):

neighbours = []

for delta in [-0.1, 0.1]:

neighbours.append(solution + delta)

return neighbours

# Find the best neighbour

def get\_best\_neighbour(neighbours):

best\_neighbour = neighbours[0]

best\_quality = objective\_function(best\_neighbour)

for neighbour in neighbours[1:]:

neighbour\_quality = objective\_function(neighbour)

if neighbour\_quality > best\_quality:

best\_quality = neighbour\_quality

best\_neighbour = neighbour

return best\_neighbour

# Hill climbing algorithm

def hill\_climbing():

current\_solution = generate\_initial\_solution()

while True:

neighbours = generate\_neighbours(current\_solution)

best\_neighbour = get\_best\_neighbour(neighbours)

if objective\_function(best\_neighbour) <= objective\_function(current\_solution):

return current\_solution

current\_solution = best\_neighbour

# Main function

def main():

best\_solution = hill\_climbing()

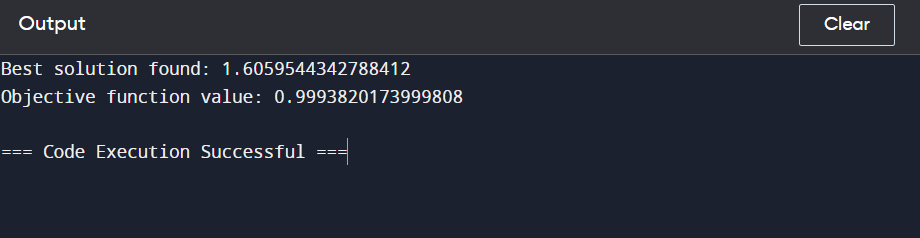
print("Best solution found:", best\_solution)

print("Objective function value:", objective\_function(best\_solution))

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

main()

****

**f(x) = -5x2+3x+6**

import random

# Define the objective function

def objective\_function(x):

return -(5 \* x \*\* 2 + 3 \* x + 6)

# Generate initial solution

def generate\_initial\_solution():

return random.uniform(-100, 100)

# Generate neighbour solutions

def generate\_neighbours(solution):

neighbours = []

for delta in [-0.1, 0.1]:

neighbours.append(solution + delta)

return neighbours

# Find the best neighbour

def get\_best\_neighbour(neighbours):

best\_neighbour = neighbours[0]

best\_quality = objective\_function(best\_neighbour)

for neighbour in neighbours[1:]:

neighbour\_quality = objective\_function(neighbour)

if neighbour\_quality > best\_quality:

best\_quality = neighbour\_quality

best\_neighbour = neighbour

return best\_neighbour

# Hill climbing algorithm

def hill\_climbing():

current\_solution = generate\_initial\_solution()

while True:

neighbours = generate\_neighbours(current\_solution)

best\_neighbour = get\_best\_neighbour(neighbours)

if objective\_function(best\_neighbour) <= objective\_function(current\_solution):

return current\_solution

current\_solution = best\_neighbour

# Main function

def main():

best\_solution = hill\_climbing()

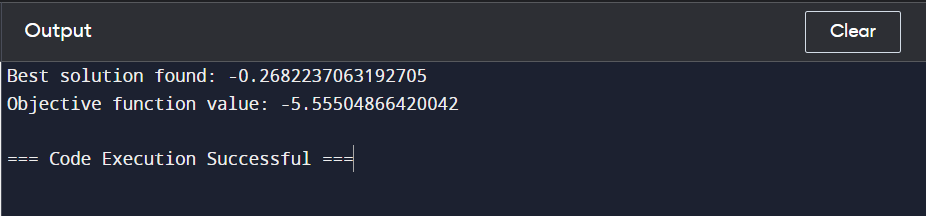
print("Best solution found:", best\_solution)

print("Objective function value:", objective\_function(best\_solution))

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

main()



1. **Genetic Algorithm**

import random

# Step 1: Determine the number of chromosomes, generation, mutation rate, and crossover rate value

def genetic\_algorithm(num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables):

# Step 2: Generate chromosome population with random initialization

population = generate\_population(num\_chromosomes, num\_variables)

i = int(1)

while True:

# Step 3: Evaluation of fitness value of chromosomes

fitness\_values = [fitness\_function(chromosome) for chromosome in population]

# Step 8: Solution (Best Chromosomes)

best\_chromosome = population[fitness\_values.index(min(fitness\_values))]

# print(f"Generation {i}: solution: {best\_chromosome}, Fitness: {min(fitness\_values)}")

if min(fitness\_values) == 0:

print(f"\nGeneration {i}: Best solution: {best\_chromosome}, Fitness: {min(fitness\_values)}")

print(f"Optimal Solution after {i} Generations")

break

i = i + 1

next\_generation = []

# Step 5: Chromosomes selection

selected\_population = selection(population, fitness\_values)

for \_ in range(num\_chromosomes // 2):

parent1, parent2 = selected\_population[random.randint(0, len(selected\_population) - 1)], selected\_population[random.randint(0, len(selected\_population) - 1)]

# Step 6: Crossover

child1, child2 = crossover(parent1, parent2, crossover\_rate)

# Step 7: Mutation

child1 = mutation(child1, mutation\_rate)

child2 = mutation(child2, mutation\_rate)

next\_generation.extend([child1, child2])

population = next\_generation

return best\_chromosome

# Step 2: Generate initial population

def generate\_population(size, num\_variables):

population = []

for \_ in range(size):

individual = [random.randint(0, 10) for \_ in range(num\_variables)]

population.append(individual)

return population

# Step 4: Fitness function evaluation

def fitness\_function(variables):

x, y, z, w = variables

return abs((a + 2\*b + 3\*c + 4\*d) - 30)

# Step 5: Chromosome selection using roulette wheel selection

def selection(population, fitness\_values):

selected\_population = []

total\_fitness = sum(fitness\_values)

probabilities = [fitness / total\_fitness for fitness in fitness\_values]

for \_ in range(len(population)):

selected = random.choices(population, probabilities)[0]

selected\_population.append(selected)

return selected\_population

# Step 6: Crossover

def crossover(parent1, parent2, crossover\_rate):

if random.random() < crossover\_rate:

crossover\_point = random.randint(1, len(parent1) - 1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

return child1, child2

return parent1, parent2

# Step 7: Mutation

def mutation(individual, mutation\_rate):

if random.random() < mutation\_rate:

index = random.randint(0, len(individual) - 1)

new\_value = random.randint(0, 10)

individual[index] = new\_value

return individual

def get\_input():

num\_chromosomes = int(input("Enter the number of chromosomes: "))

mutation\_rate = float(input("Enter the mutation rate: "))

crossover\_rate = float(input("Enter the crossover rate: "))

num\_variables = 4 # Assuming 4 variables (a,b,c,d) for the given equation

return num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables

def main():

num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables = get\_input()

best\_chromosome = genetic\_algorithm(num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables)

print("Best solution:", best\_chromosome)

if \_\_name\_\_ == "\_\_main\_\_":

main()

