PRACTICAL NO: 7

AIM: Implement perceptron algorithm for AND/OR

CODE:

# TYCS\_AMAN\_523

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

if v >= 0:

return 1

else:

return 0

# design Perceptron Model

def perceptronModel(x, w, b):

v = np.dot(w, x) + b

y = unitStep(v)

return y

# AND Logic Function

# w1 = 1, w2 = 1, b = -1.5

def AND\_logicFunction(x):

w = np.array([1, 1])

b = -1.5

return perceptronModel(x, w, b)

# OR Logic Function

# w1 = 1, w2 = 1, b = -0.5

def OR\_logicFunction(x):

w = np.array([1, 1])

b = -0.5

return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("AND({}, {}) = {}".format(0, 1, AND\_logicFunction(test1)))

print("AND({}, {}) = {}".format(1, 1, AND\_logicFunction(test2)))

print("AND({}, {}) = {}".format(0, 0, AND\_logicFunction(test3)))

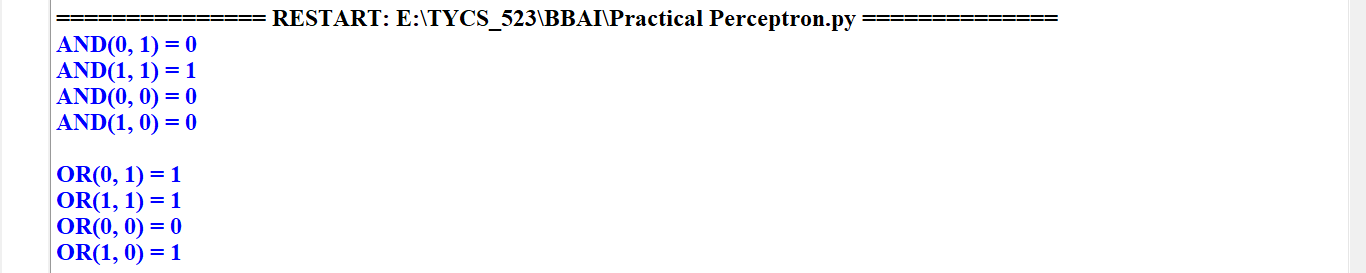
print("AND({}, {}) = {}".format(1, 0, AND\_logicFunction(test4)))

print("\nOR({}, {}) = {}".format(0, 1, OR\_logicFunction(test1)))

print("OR({}, {}) = {}".format(1, 1, OR\_logicFunction(test2)))

print("OR({}, {}) = {}".format(0, 0, OR\_logicFunction(test3)))

print("OR({}, {}) = {}".format(1, 0, OR\_logicFunction(test4)))

OUTPUT:  


PRACTICAL NO: 6A

AIM: Naïve bayes classifier (TITANIC)

CODE:

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)

from sklearn.model\_selection import train\_test\_split

df=pd.read\_csv("E:\\TYCS\_523\\BBAI\\train\_clean.csv")

df\_test=pd.read\_csv("E:\\TYCS\_523\\BBAI\\test\_clean.csv")

df.info() # Print a concise summary of a DataFrame.

df.head() # Return the first 5 rows.

df.columns # Return the column labels of the DataFrame.

df.describe() # Generate descriptive statistics.

# Since majority of cabin values are missing -> remove the column

# \* PassengerId is unique -> drop column

# \* Name is unique -> drop column

# \* TicketId is unique-> drop column

# \* They do not contribute to the survival probability.

df.drop(["Cabin","Name","PassengerId","Ticket"],axis=1,inplace=True)

df\_test.drop(["Cabin","Name","PassengerId","Ticket"],axis=1,inplace=True)

df=df[['Age', 'Embarked', 'Fare', 'Parch', 'Pclass', 'Sex', 'SibSp', 'Title', 'Family\_Size','Survived']]

df\_test=df\_test[['Age', 'Embarked', 'Fare', 'Parch', 'Pclass', 'Sex', 'SibSp', 'Title', 'Family\_Size','Survived']]

# Age and fare are in float -> convert it into integer and then into categories.

# Age is grouped into 7 categories

data=[df,df\_test]

for d in data:

d['Age'] = d['Age'].astype(int)

d.loc[ d['Age'] <= 10, 'Age'] = 0

d.loc[(d['Age'] > 10) & (d['Age'] <= 18), 'Age'] = 1

d.loc[(d['Age'] > 18) & (d['Age'] <= 25), 'Age'] = 2

d.loc[(d['Age'] > 25) & (d['Age'] <= 30), 'Age'] = 3

d.loc[(d['Age'] > 30) & (d['Age'] <= 35), 'Age'] = 4

d.loc[(d['Age'] > 35) & (d['Age'] <= 40), 'Age'] = 5

d.loc[(d['Age'] > 40) & (d['Age'] <= 65), 'Age'] = 6

d.loc[ d['Age'] > 65, 'Age'] = 6

# Fare is grouped into 6 categories

data = [df,df\_test]

for d in data:

d.loc[ d['Fare'] <= 8, 'Fare'] = 0

d.loc[(d['Fare'] > 8) & (d['Fare'] <= 15), 'Fare'] = 1

d.loc[(d['Fare'] > 15) & (d['Fare'] <= 31), 'Fare'] = 2

d.loc[(d['Fare'] > 31) & (d['Fare'] <= 99), 'Fare'] = 3

d.loc[(d['Fare'] > 99) & (d['Fare'] <= 250), 'Fare'] = 4

d.loc[ d['Fare'] > 250, 'Fare'] = 5

d['Fare'] = d['Fare'].astype(int)

# Convert Survived from float-> int

df.Survived=df.Survived.astype(int)

# Creating test and training samples. Splitting the dataframe into two random samples(80% and 20%) for traing and testing.

train, test = train\_test\_split(df, test\_size=0.2)

survived\_yes=train.loc[train.Survived==1]

P\_yes=len(survived\_yes)/len(train)

P\_yes # Probability of Survival in training data

survived\_no=train.loc[train.Survived==0]

P\_no=len(survived\_no)/len(train)

P\_no # Probability of not Survival in training data

# value counts of each category of an attribute.

for col in train.columns:

count=train[col].value\_counts()

print(count)

atr=list(df.columns.values)

output\_dataframe= pd.DataFrame(columns = ['Actual', 'Predicted'])

for i in test.itertuples():

test1=list(i)

test1.pop(0) # removing Index (unwanted)

ans=test1.pop() # removing actual value

py=1

for i in range(9):

val = train[(train[atr[i]] == test1[i]) & (train.Survived == 1)].count().values.item(0)

py = py \* (val) / len(survived\_yes)

total\_yes = py \* P\_yes

pn=1

for i in range(9):

val = train[(train[atr[i]] == test1[i]) & (train.Survived == 0)].count().values.item(0)

pn = pn \* (val) / len(survived\_no)

total\_no = pn \* P\_no

if total\_yes>total\_no:

list1=[[ans,1]] #Survived

output\_dataframe = pd.concat([output\_dataframe, pd.DataFrame(list1, columns=['Actual', 'Predicted'])], ignore\_index=True)

else:

list0=[[ans,0]] #NotSurvived

output\_dataframe = pd.concat([output\_dataframe, pd.DataFrame(list0, columns=['Actual', 'Predicted'])], ignore\_index=True)

# Evaluation metrics

TP=0

TN=0

FP=0

FN=0

for index,row in output\_dataframe.iterrows():

if row['Predicted']== row['Actual'] and row['Predicted']==1:

TP += 1

elif row['Predicted']== row['Actual'] and row['Predicted']==0:

TN +=1

elif row['Predicted']==1:

FP +=1

else:

FN +=1

# Accuracy = [TP + TN] / Total Population

accuracy= (TP+TN)/len(output\_dataframe)

print("The accuracy for the test set is ",accuracy \*100,"%")

# Precision = TP / [TP + FP]

# tells us about the success probability of making a correct positive class classification.

precision = TP / (TP+FP)

print("The precision for the test set is ",precision \*100,"%")

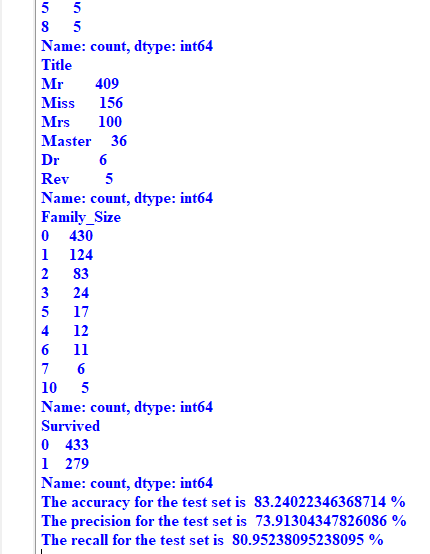
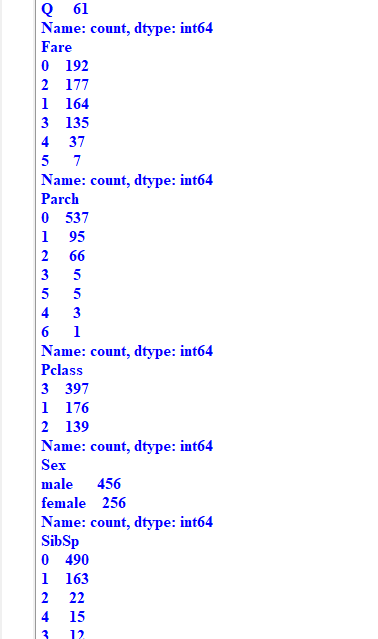
# Recall = TP / [TP + FN]

# explains how sensitive the model is towards identifying the positive class.

recall = TP / (TP+FN)

print("The recall for the test set is ",recall \*100,"%")

OUTPUT:



PRACTICAL NO: 6B

AIM: WEATHER

CODE:

#TYCS\_AMAN\_523

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

weather = ['sunnny','sunny','overcast','rainy','rainy','rainy','overcast','sunny','sunny','rainy','sunny','overcast','overcast','rainy']

temp = ['hot','hot','hot','mild','cool','cool','cool','mild','cool','mild','mild','mild','hot','mild']

play = ['no','no','yes','yes','yes','no','yes','no','yes','yes','yes','yes','yes','no']

from sklearn import preprocessing

le = preprocessing.LabelEncoder()

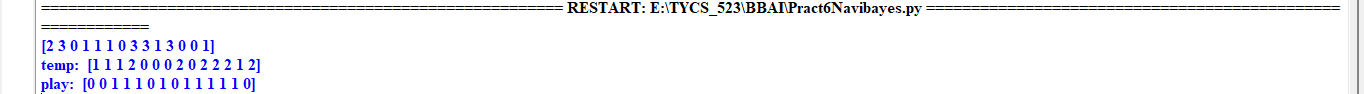
weather\_encode = le.fit\_transform(weather)

print(weather\_encode)

temp\_encode = le.fit\_transform(temp)

label = le.fit\_transform(play)

print("temp: ",temp\_encode)

print("play: ",label)  
  
OUTPUT:  


PRACTICAL NO: 5

AIM: Bayesian classification (Heart Disease)

CODE:

import pandas as pd

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianNetwork

from pgmpy.inference import VariableElimination

# Load the data

data = pd.read\_csv("E:\\TYCS\_523\\BBAI\\ds4.csv")

heart\_disease = pd.DataFrame(data)

#print(heart\_disease)

# Define the Bayesian network structure

model = BayesianNetwork([

('age', 'Lifestyle'),

('Gender', 'Lifestyle'),

('Family', 'heartdisease'),

('diet', 'cholestrol'),

('Lifestyle', 'diet'),

('cholestrol', 'heartdisease'),

])

# Fit the model

model.fit(heart\_disease, estimator=MaximumLikelihoodEstimator)

# Create a Variable Elimination object

HeartDisease\_infer = VariableElimination(model)

# Get user input for each variable

age = int(input('Enter Age (SuperSenior Citizen:0, SeniorCitizen:1, MiddleAged:2, Youth:3, Teen:4): '))

gender = int(input('Enter Gender (Male:0, Female:1): '))

family\_history = int(input('Enter Family History (Yes:1, No:0): '))

diet = int(input('Enter Diet (High:0, Medium:1): '))

lifestyle = int(input('Enter Lifestyle (Athlete:0, Active:1, Moderate:2, Sedentary:3): '))

cholesterol = int(input('Enter Cholesterol (High:0, BorderLine:1, Normal:2): '))

# Query the model for the probability of heart disease

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={

'age': age,

'Gender': gender,

'Family': family\_history,

'diet': diet,

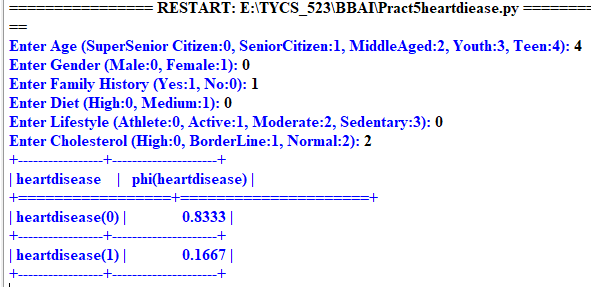
'Lifestyle': lifestyle,

'cholestrol': cholesterol

})

# Print the result

print(q)  
  
OUTPUT:



PRACTICAL NO: 4

AIM: Decision Tree

1. IRIS DATASET

CODE:

OUTPUT:

1. DIABETES DATASET

CODE:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.metrics import accuracy\_score

# Load your data (assuming you're using 'diabetes.csv' or another dataset)

# If needed, adjust the target variable and feature columns accordingly

df = pd.read\_csv("E://TYCS\_523//BBAI//diabetes.csv")

# Assuming you want to predict a binary target column (e.g., 'Outcome' or 'target')

# Split data into features (X) and target (y)

X = df.drop(columns='Outcome') # Adjust 'Outcome' if your target column is different

y = df['Outcome'] # Adjust the target column name if necessary

# Split data into training and testing sets

X\_train\_bopt, X\_test\_bopt, y\_train\_bopt, y\_test\_bopt = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create and fit decision tree classifier without max depth limit

tree = DecisionTreeClassifier(random\_state=1)

tree.fit(X\_train\_bopt, y\_train\_bopt)

# Print information about the model

print(f'Decision Tree has {tree.tree\_.node\_count} nodes with a maximum depth of {tree.tree\_.max\_depth}.')

print(f'Model Accuracy for train data: {tree.score(X\_train\_bopt, y\_train\_bopt)}')

print(f'Model Accuracy for test data: {tree.score(X\_test\_bopt, y\_test\_bopt)}')

# Create and fit decision tree with maximum depth of 3

tree = DecisionTreeClassifier(max\_depth=3, random\_state=1)

tree.fit(X\_train\_bopt, y\_train\_bopt)

# Plot the decision tree

plt.figure(figsize=(25, 10)) # Adjust the figure size to make it larger

plot\_tree(tree, feature\_names=X.columns,

class\_names=['<=50K', '>50K'], # Adjust according to your classes

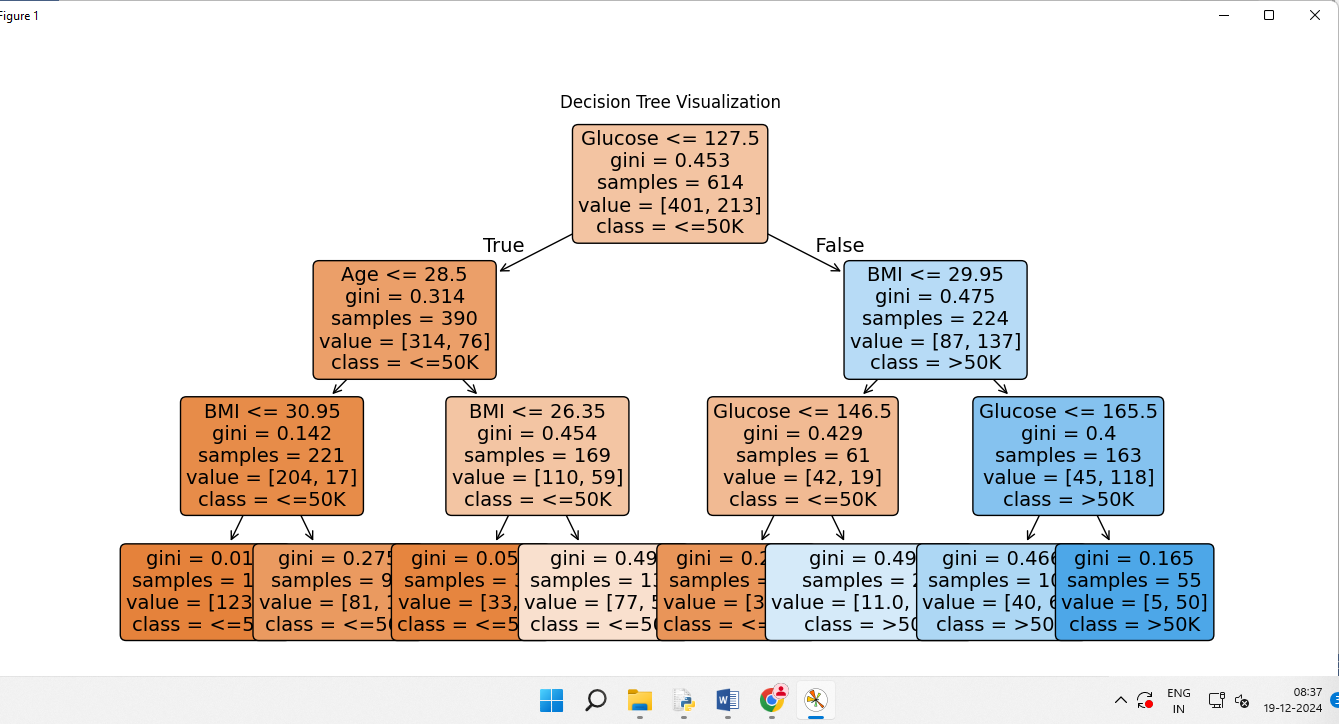
filled=True, rounded=True, fontsize=14)

# Display the plot

plt.title("Decision Tree Visualization")

plt.show()

OUTPUT:



1. INCOME DATA – RANDOM FOREST

CODE:

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.metrics import accuracy\_score

# Step 1: Load and preprocess the data

# Assign proper column names for the dataset

column\_names = [

"age", "workclass", "fnlwgt", "education", "education-num",

"marital-status", "occupation", "relationship", "race", "sex",

"capital-gain", "capital-loss", "hours-per-week", "native-country", "income"

]

# Load the dataset with column names

file\_path = "E://TYCS\_523//BBAI//adultdata.csv"

df = pd.read\_csv(file\_path, names=column\_names, skipinitialspace=True)

# Encode categorical features using one-hot encoding

categorical\_features = [

"workclass", "education", "marital-status", "occupation",

"relationship", "race", "sex", "native-country"

]

df = pd.get\_dummies(df, columns=categorical\_features)

# Encode the target variable: income (binary classification)

df["income"] = df["income"].apply(lambda x: 1 if x == ">50K" else 0)

# Step 2: Split the data into training and testing sets

X = df.drop(columns=["income"]) # Features

y = df["income"] # Target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Step 3: Train the decision tree classifier

# Use a limited depth for readability in visualization

tree = DecisionTreeClassifier(max\_depth=3, random\_state=1)

tree.fit(X\_train, y\_train)

# Step 4: Evaluate the model

train\_accuracy = accuracy\_score(y\_train, tree.predict(X\_train))

test\_accuracy = accuracy\_score(y\_test, tree.predict(X\_test))

print(f"Training Accuracy: {train\_accuracy:.2f}")

print(f"Testing Accuracy: {test\_accuracy:.2f}")

# Step 5: Visualize the decision tree

plt.figure(figsize=(20, 10))

plot\_tree(

tree,

feature\_names=X.columns, # Feature names for clarity

class\_names=["<=50K", ">50K"], # Target class names

filled=True, # Color the nodes for better visual distinction

rounded=True, # Rounded corners for better readability

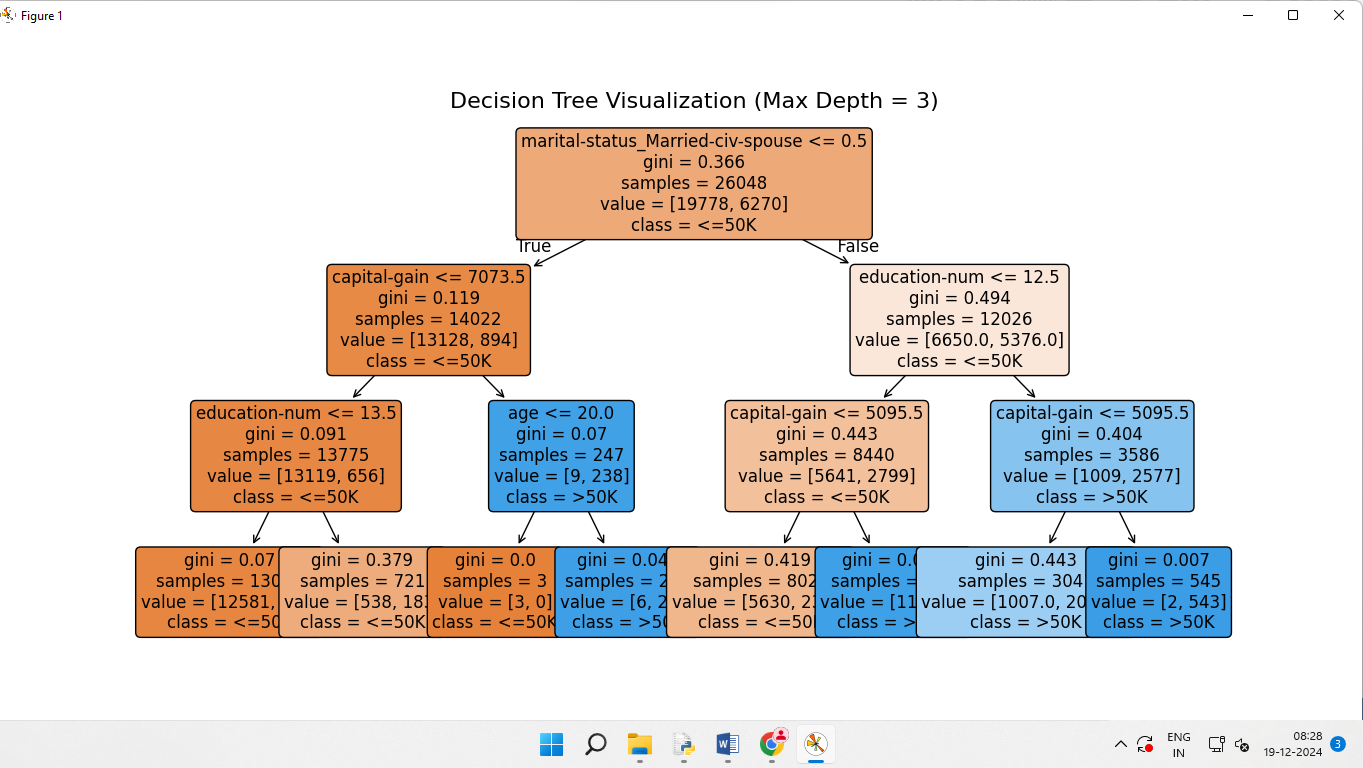
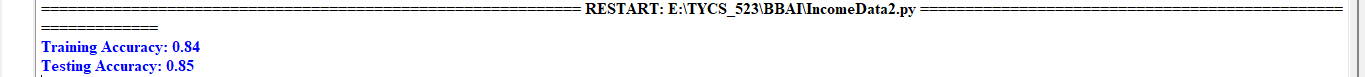
fontsize=12 # Larger font size for labels

)

plt.title("Decision Tree Visualization (Max Depth = 3)", fontsize=16)

plt.show()

OUTPUT:



PRACTICAL NO: 3

AIM: Heuristic search algorithms

1. Hill Climbing Search

CODE:

from math import inf

class Node:

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

self.state = state # The state of the node (could be any representation)

self.value = value # The evaluation function (heuristic) value of the node

def get\_children(self):

"""Returns the list of neighboring nodes (children). For demonstration,

we return nodes that are neighbors in the search space."""

children = []

if self.state > 0:

children.append(Node(self.state - 1, self.value - 1)) # Decreasing state

if self.state < 10:

children.append(Node(self.state + 1, self.value + 1)) # Increasing state

return children

def evaluate(node: Node) -> float:

"""Evaluate the node based on some criteria (could be a heuristic)."""

return node.value # Simple evaluation based on value, can be modified

def hill\_climbing(start\_node: Node) -> Node:

"""

Hill Climbing Algorithm - finds a local maximum starting from the start node.

:param start\_node: The starting node for the search.

:return: The best node found by hill climbing.

"""

current\_node = start\_node

best\_value = -inf

best\_node = None

print(f"Starting Hill Climbing from node {current\_node.state} with value {current\_node.value}")

while True:

# Get the value for the current node

current\_value = evaluate(current\_node)

print(f"Evaluating node {current\_node.state}: {current\_value}")

# Check if the current node is better than the best node so far

if current\_value > best\_value:

best\_node = current\_node

best\_value = current\_value

else:

# If no better node was found, stop and return the best node found

print("No better neighbor found, returning the best node.")

return best\_node

# Get all the children (neighbors) of the current node

children = current\_node.get\_children()

# Explore all children and pick the one with the best value

for child in children:

child\_value = evaluate(child)

print(f"Evaluating child node {child.state}: {child\_value}")

if child\_value > best\_value:

best\_value = child\_value

best\_node = child

current\_node = child # Move to this child node

print(f"Moving to better node {child.state} with value {child\_value}")

break

else:

# If no child is better than the current, stop the search

print("No better child found, returning the best node.")

return best\_node

# Example of usage:

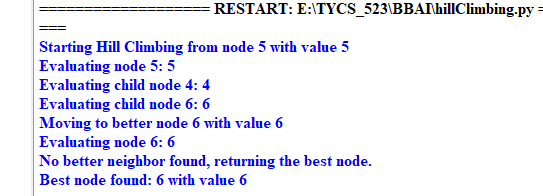
start\_node = Node(state=5, value=5) # Starting node with state 5 and value 5

result\_node = hill\_climbing(start\_node)

# Output the result

print(f"Best node found: {result\_node.state} with value {result\_node.value}")

OUTPUT:



1. Min Max Algorithms

CODE:

MAX, MIN = float('inf'), float('-inf')

def alpha\_beta(depth, node\_index, maximizing\_player, values, alpha, beta):

# Terminating condition. i.e # leaf node is reached

if depth == 0:

return values[node\_index]

if maximizing\_player:

best = MIN

# Recur for left and right children

for i in range(0, 2):

val = alpha\_beta(depth - 1, node\_index \* 2 + i, False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

else:

best = MAX

# Recur for left and # right children

for i in range(0, 2):

val = alpha\_beta(depth - 1, node\_index \* 2 + i, True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

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

values = [3, 4, 7, 8, 1, 2, 0, -1]

print("The optimal value is :", alpha\_beta(3, 0, True, values, MIN, MAX))

OUTPUT:



1. Constraint Satisfaction Algorithms

CODE:

# Variables and Constraints for the CSP problem

VARIABLES = ["A", "B", "C", "D", "E", "F", "G"]

CONSTRAINTS = [

("A", "B"), ("A", "C"), ("B", "C"), ("B", "D"), ("B", "E"),

("C", "E"), ("C", "F"), ("D", "E"), ("E", "F"), ("E", "G"), ("F", "G")

]

# Backtracking search function

def backtrack(assignment):

"""Runs backtracking search to find an assignment."""

# Check if assignment is complete (all variables assigned)

if len(assignment) == len(VARIABLES):

return assignment

# Select an unassigned variable

var = select\_unassigned\_variable(assignment)

# Try all possible values for the variable

for value in ["Monday", "Tuesday", "Wednesday"]:

new\_assignment = assignment.copy()

new\_assignment[var] = value

# If the new assignment is consistent, recursively call backtrack

if consistent(new\_assignment):

result = backtrack(new\_assignment)

if result is not None:

return result

return None # If no solution found

# Function to select the next unassigned variable (returns the first unassigned variable)

def select\_unassigned\_variable(assignment):

"""Chooses a variable not yet assigned, in order."""

for variable in VARIABLES:

if variable not in assignment:

return variable

return None # If all variables are assigned

# Function to check if the current assignment is consistent with the constraints

def consistent(assignment):

"""Checks to see if an assignment is consistent."""

# Check each constraint

for (x, y) in CONSTRAINTS:

# Only consider arcs where both variables are assigned

if x in assignment and y in assignment:

# If both have the same value, the assignment is inconsistent

if assignment[x] == assignment[y]:

return False

return True # If no conflicts, assignment is consistent

# Perform backtracking search and print the solution

solution = backtrack(dict())

print(solution)

OUTPUT:



PRACTICAL NO: 2

**AIM: Informed Search Algorithms**

1. **Greedy Best-First Search**

**CODE:**

**import heapq**

**import networkx as nx**

**import matplotlib.pyplot as plt**

**# Node class to store information about each node**

**class Node:**

**def \_\_init\_\_(self, name, heuristic):**

**self.name = name**

**self.heuristic = heuristic**

**def \_\_lt\_\_(self, other):**

**return self.heuristic < other.heuristic**

**# Greedy Best-First Search for Hierarchical Routing**

**def greedy\_best\_first\_search\_hierarchical(graph, start, goal, heuristic, region\_map):**

**# Priority queue to explore nodes, sorted by heuristic**

**priority\_queue = []**

**heapq.heappush(priority\_queue, Node(start, heuristic[start]))**

**visited = set() # To track visited nodes**

**path = {start: None} # Track paths**

**while priority\_queue:**

**current\_node = heapq.heappop(priority\_queue).name**

**# Goal reached, reconstruct path**

**if current\_node == goal:**

**return reconstruct\_path(path, start, goal)**

**visited.add(current\_node)**

**# Explore neighbors in the same region first**

**current\_region = region\_map[current\_node]**

**for neighbor in graph[current\_node]:**

**if neighbor not in visited and region\_map[neighbor] == current\_region:**

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

**if neighbor not in path:**

**path[neighbor] = current\_node**

**# Explore neighbors in other regions**

**for neighbor in graph[current\_node]:**

**if neighbor not in visited and region\_map[neighbor] != current\_region:**

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

**if neighbor not in path:**

**path[neighbor] = current\_node**

**return None # Return None if no path is found**

**# Helper function to reconstruct the path from start to goal**

**def reconstruct\_path(path, start, goal):**

**current = goal**

**result\_path = []**

**while current is not None:**

**result\_path.append(current)**

**current = path[current]**

**result\_path.reverse()**

**return result\_path**

**# Function to calculate the total cost of the path**

**def calculate\_path\_cost(path, heuristic):**

**return sum(heuristic[node] for node in path)**

**# Function to visualize the graph and the path**

**def visualize\_graph(graph, path, pos):**

**G = nx.DiGraph() # Directed graph to show arrows**

**# Add edges to the graph**

**for node, neighbors in graph.items():**

**for neighbor in neighbors:**

**G.add\_edge(node, neighbor)**

**# Plot the graph**

**plt.figure(figsize=(10, 8))**

**# Draw nodes and edges**

**nx.draw(**

**G, pos, with\_labels=True, node\_size=4000, node\_color='skyblue',**

**font\_size=15, font\_weight='bold', edge\_color='gray', arrows=True**

**)**

**# Highlight the path**

**if path:**

**path\_edges = list(zip(path, path[1:]))**

**nx.draw\_networkx\_edges(**

**G, pos, edgelist=path\_edges, edge\_color='green',**

**width=3, arrows=True, arrowstyle='-|>'**

**)**

**nx.draw\_networkx\_nodes(G, pos, nodelist=path, node\_color='lightgreen')**

**# Visual cues for interpretation**

**plt.title("Greedy Best-First Search", size=20)**

**plt.show()**

**# Graph definition**

**graph = {**

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

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

**'C': ['F', 'G'],**

**'E': ['H'],**

**'F': ['H'],**

**'G': ['H'],**

**'D': []**

**}**

**# Heuristic values (estimate of cost to goal)**

**heuristic = {**

**'A': 13,**

**'B': 12,**

**'C': 4,**

**'D': 7,**

**'E': 3,**

**'F': 8,**

**'G': 2,**

**'H': 0**

**}**

**# Regions for hierarchical routing**

**region\_map = {**

**'A': 1, 'B': 1, 'C': 1,**

**'D': 2, 'E': 2,**

**'F': 3, 'G': 3,**

**'H': 3**

**}**

**# Node positions for layout (hierarchical with 'A' at the top)**

**pos = {**

**'A': (0, 3),**

**'B': (-1, 2),**

**'C': (1, 2),**

**'D': (-1.5, 1),**

**'E': (-0.5, 1),**

**'F': (0.5, 1),**

**'G': (1.5, 1),**

**'H': (0, 0)**

**}**

**# Perform Greedy Best-First Search for hierarchical routing**

**start\_node = 'A'**

**goal\_node = 'H'**

**result\_path = greedy\_best\_first\_search\_hierarchical(graph, start\_node, goal\_node, heuristic, region\_map)**

**# Print the path and its total cost**

**if result\_path:**

**print("Greedy Best-First Search")**

**print(f"Path from {start\_node} to {goal\_node}: {result\_path}")**

**path\_cost = calculate\_path\_cost(result\_path, heuristic)**

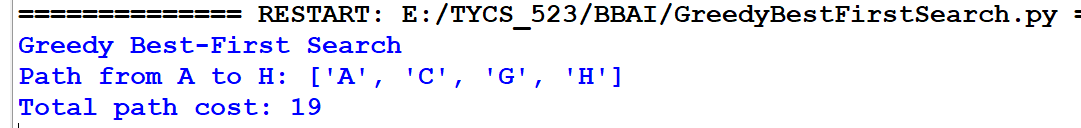
**print(f"Total path cost: {path\_cost}")**

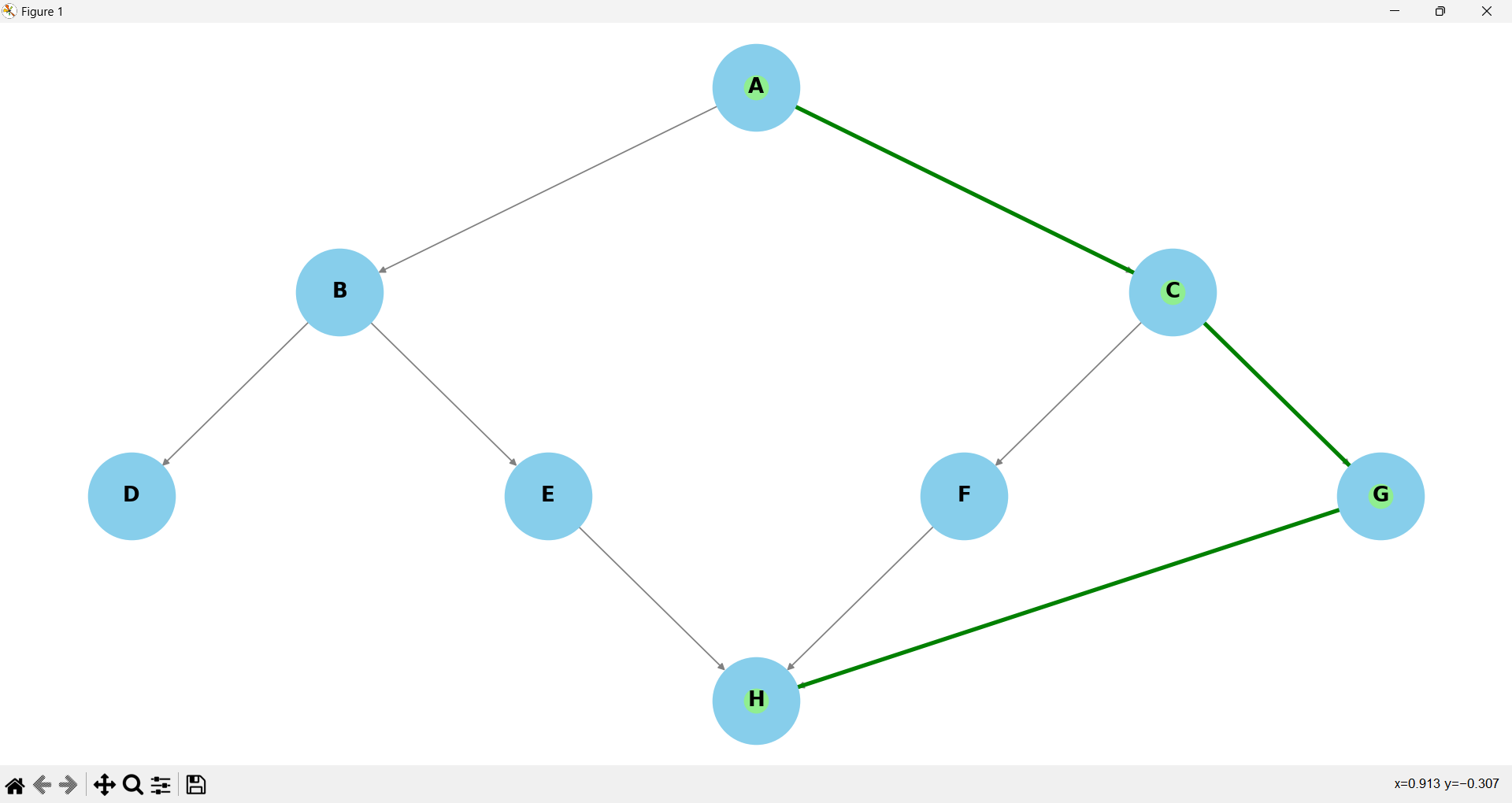
**else:**

**print("No path found.")**

**# Visualize the graph and the found path**

**visualize\_graph(graph, result\_path, pos)**

**OUTPUT: **



1. **A\* Algorithm**

**CODE:**

**#TYCS\_AMAN\_523**

**import heapq**

**import networkx as nx**

**import matplotlib.pyplot as plt**

# Node class to store information about each node

**class Node:**

**def \_\_init\_\_(self, name, g, h):**

**self.name = name**

**self.g = g** # Cost from start to node

**self.h = h** # Heuristic to goal

**self.f = g + h** # f = g + h (total estimated cost)

**def \_\_lt\_\_(self, other):**

**return self.f < other.f**

# A\* Search Algorithm

**def a\_star\_search(graph, start, goal, heuristic):**

# Priority queue (min-heap) to explore nodes with the lowest f(n)

**open\_list = []**

**heapq.heappush(open\_list, Node(start, 0, heuristic[start]))**

# To track the explored nodes

**explored = set()**

# To reconstruct the path

**path = {start: None}**

# Dictionary to store the g values for each node

**g\_values = {start: 0}**

**while open\_list:**

**current\_node = heapq.heappop(open\_list).name**

# If goal is reached, reconstruct the path

**if current\_node == goal:**

**return reconstruct\_path(path, start, goal)**

**explored.add(current\_node)**

# Explore the neighbors

**for neighbor in graph[current\_node]:**

**if neighbor in explored:**

**continue**

**g\_new = g\_values[current\_node] + 1** # Each edge has a cost of 1

**h\_new = heuristic[neighbor]**

**f\_new = g\_new + h\_new**

# If neighbor not in g\_values or found a better path

**if neighbor not in g\_values or g\_values[neighbor] > g\_new:**

**g\_values[neighbor] = g\_new**

**heapq.heappush(open\_list, Node(neighbor, g\_new, h\_new))**

**path[neighbor] = current\_node**

**return None # If no path is found**

# Helper function to reconstruct the path from start to goal

**def reconstruct\_path(path, start, goal):**

**current = goal**

**result\_path = []**

**while current is not None:**

**result\_path.append(current)**

**current = path[current]**

**result\_path.reverse()**

**return result\_path**

# Function to visualize the graph and the path

**def visualize\_graph(graph, path, pos):**

**G = nx.DiGraph()** # Use a directed graph to show arrows

# Add edges to the graph

**for node, neighbors in graph.items():**

**for neighbor in neighbors:**

**G.add\_edge(node, neighbor)**

# Plot the graph

**plt.figure(figsize=(10, 8))**

# Draw the nodes and edges

**nx.draw(**

**G, pos, with\_labels=True, node\_size=4000, node\_color='skyblue',**

**font\_size=15, font\_weight='bold', edge\_color='gray', arrows=True**

**)**

# Highlight the path

**if path:**

**path\_edges = list(zip(path, path[1:]))**

**nx.draw\_networkx\_edges(**

**G, pos, edgelist=path\_edges, edge\_color='green',**

**width=3, arrows=True, arrowstyle='-|>'**

**)**

**nx.draw\_networkx\_nodes(G, pos, nodelist=path, node\_color='lightgreen')**

**plt.title("A\* Search", size=20)**

**plt.show()**

# Helper function to calculate the path cost

**def calculate\_path\_cost(path, heuristic):**

**return sum(heuristic[node] for node in path)**

# Graph definition

**graph = {**

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

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

**'C': ['F', 'G'],**

**'E': ['H'],**

**'F': ['H'],**

**'G': ['H'],**

**'D': []**

**}**

# Heuristic values

**heuristic = {**

**'A': 13, 'B': 12, 'C': 4, 'D': 7, 'E': 3, 'F': 8, 'G': 2, 'H': 0**

**}**

# Node positions for layout

**pos = {**

**'A': (0, 3),**

**'B': (-1, 2),**

**'C': (1, 2),**

**'D': (-1.5, 1),**

**'E': (-0.5, 1),**

**'F': (0.5, 1),**

**'G': (1.5, 1),**

**'H': (0, 0)**

**}**

# Perform A\* Search

**start\_node = 'A'**

**goal\_node = 'H'**

# Execute A\* Search for the path

**result\_path = a\_star\_search(graph, start\_node, goal\_node, heuristic)**

**if result\_path:**

**print("A\* Search")**

**print(f"Path from {start\_node} to {goal\_node}: {result\_path}")**

**path\_cost = calculate\_path\_cost(result\_path, heuristic)**

**print(f"Total path cost: {path\_cost}")**

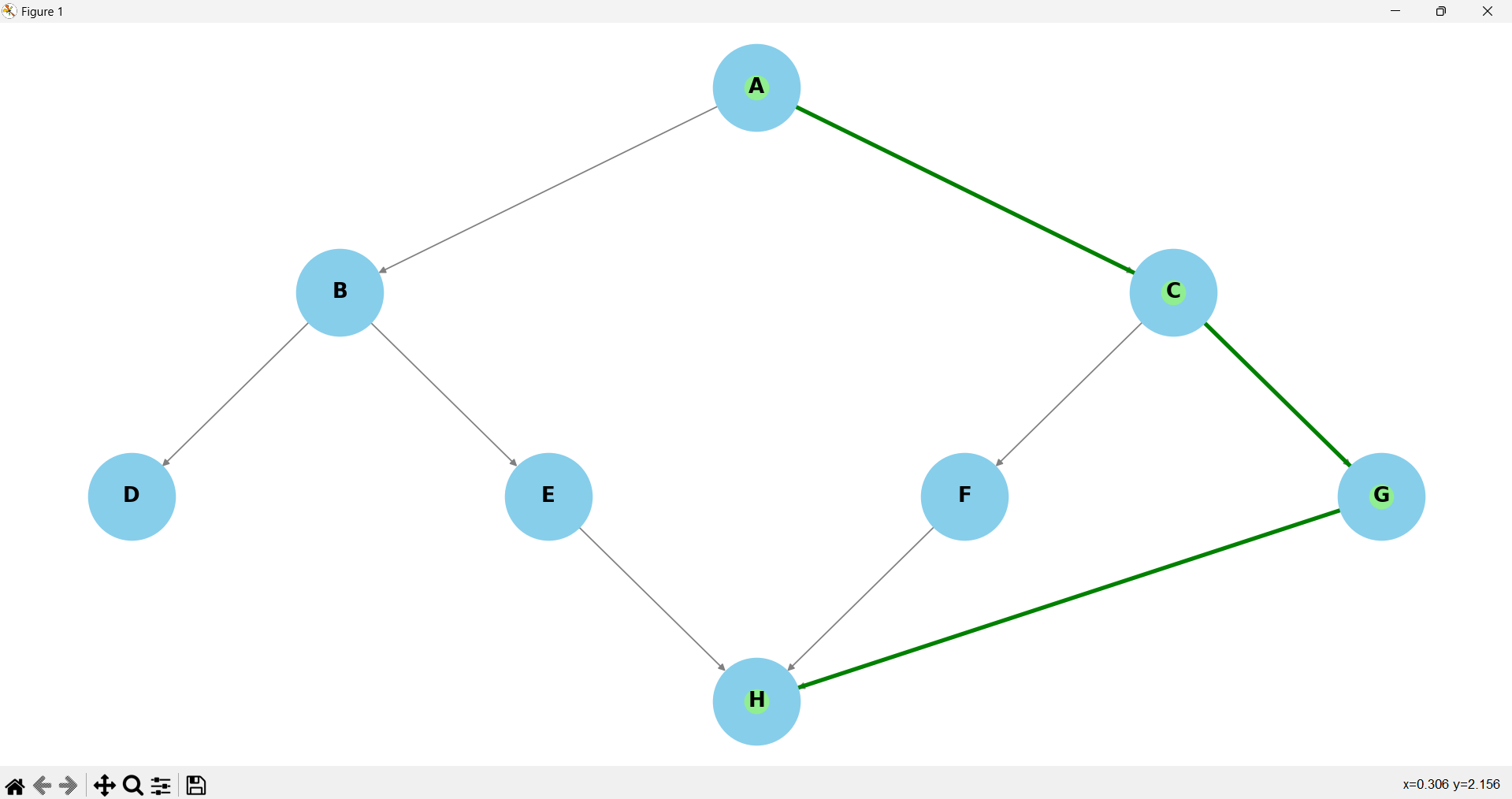
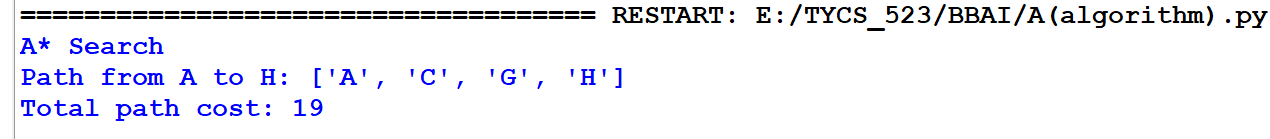
**else:**

**print("No path found.")**

# Visualize the graph and the path

**visualize\_graph(graph, result\_path, pos)**

**OUTPUT:**

****

1. **IDA (Iterative Deepening A)\*\***

**CODE:**

**#TYCS\_AMAN\_523**

**import networkx as nx**

**import matplotlib.pyplot as plt**

# Node class to store information about each node

**class Node:**

**def \_\_init\_\_(self, name, g, h):**

**self.name = name**

**self.g = g** # Cost from start to node

**self.h = h**  # Heuristic to goal

**self.f = g + h** # f = g + h (total estimated cost)

**def \_\_lt\_\_(self, other):**

**return self.f < other.f**

# IDA\* search for the graph

**def ida\_star(graph, start, goal, heuristic, max\_depth):**

**def dfs(node, g, path, depth\_limit):**

**if g + heuristic[node] > depth\_limit:** # Prune if cost exceeds limit

**return None, g**

**path.append(node)**

**if node == goal:**

**return path, g**

**min\_f = float('inf')**

**for neighbor in graph[node]:**

**if neighbor not in path:** # Avoid cycles

**new\_path, f = dfs(neighbor, g + 1, path[:], depth\_limit)**

**if new\_path:**

**return new\_path, f**

**min\_f = min(min\_f, f)**

**return None, min\_f**

**depth\_limit = heuristic[start]**

**while True:**

**path, min\_f = dfs(start, 0, [], depth\_limit)**

**if path:**

**return path**

**if min\_f == float('inf'):**

**return None # No solution found**

**depth\_limit = min\_f**

# Function to visualize the graph and the path

**def visualize\_graph(graph, path, pos):**

**G = nx.DiGraph()** # Use a directed graph to show arrows

# Add edges to the graph

**for node, neighbors in graph.items():**

**for neighbor in neighbors:**

**G.add\_edge(node, neighbor)**

# Plot the graph

**plt.figure(figsize=(10, 8))**

# Draw the nodes and edges

**nx.draw(**

**G, pos, with\_labels=True, node\_size=4000, node\_color='skyblue',**

**font\_size=15, font\_weight='bold', edge\_color='gray', arrows=True**

**)**

# Highlight the path

**if path:**

**path\_edges = list(zip(path, path[1:]))**

**nx.draw\_networkx\_edges(**

**G, pos, edgelist=path\_edges, edge\_color='green',**

**width=3, arrows=True, arrowstyle='-|>'**

**)**

**nx.draw\_networkx\_nodes(G, pos, nodelist=path, node\_color='lightgreen')**

**plt.title("IDA\*\* Search", size=20)**

**plt.show()**

# Helper function to calculate the path cost

**def calculate\_path\_cost(path, heuristic):**

**return sum(heuristic[node] for node in path)**

# Graph definition

**graph = {**

**'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F', 'G'], 'E': ['H'],'F': ['H'], 'G': ['H'],'D': []**

**}**

# Heuristic values

**heuristic = {**

**'A': 13, 'B': 12,'C': 4, 'D': 7,'E': 3, 'F': 8, 'G': 2, 'H': 0**

**}**

# Node positions for layout

**pos = {**

**'A': (0, 3), 'B': (-1, 2), 'C': (1, 2), 'D': (-1.5, 1), 'E': (-0.5, 1), 'F': (0.5, 1),**

**'G': (1.5, 1), 'H': (0, 0)**

**}**

# Perform IDA\* search

**start\_node = 'A'**

**goal\_node = 'H'**

# Execute IDA\*\* search for the path

**result\_path = ida\_star(graph, start\_node, goal\_node, heuristic, max\_depth=10)**

**if result\_path:**

**print("IDA\*\* Search")**

**print(f"Path from {start\_node} to {goal\_node}: {result\_path}")**

**path\_cost = calculate\_path\_cost(result\_path, heuristic)**

**print(f"Total path cost: {path\_cost}")**

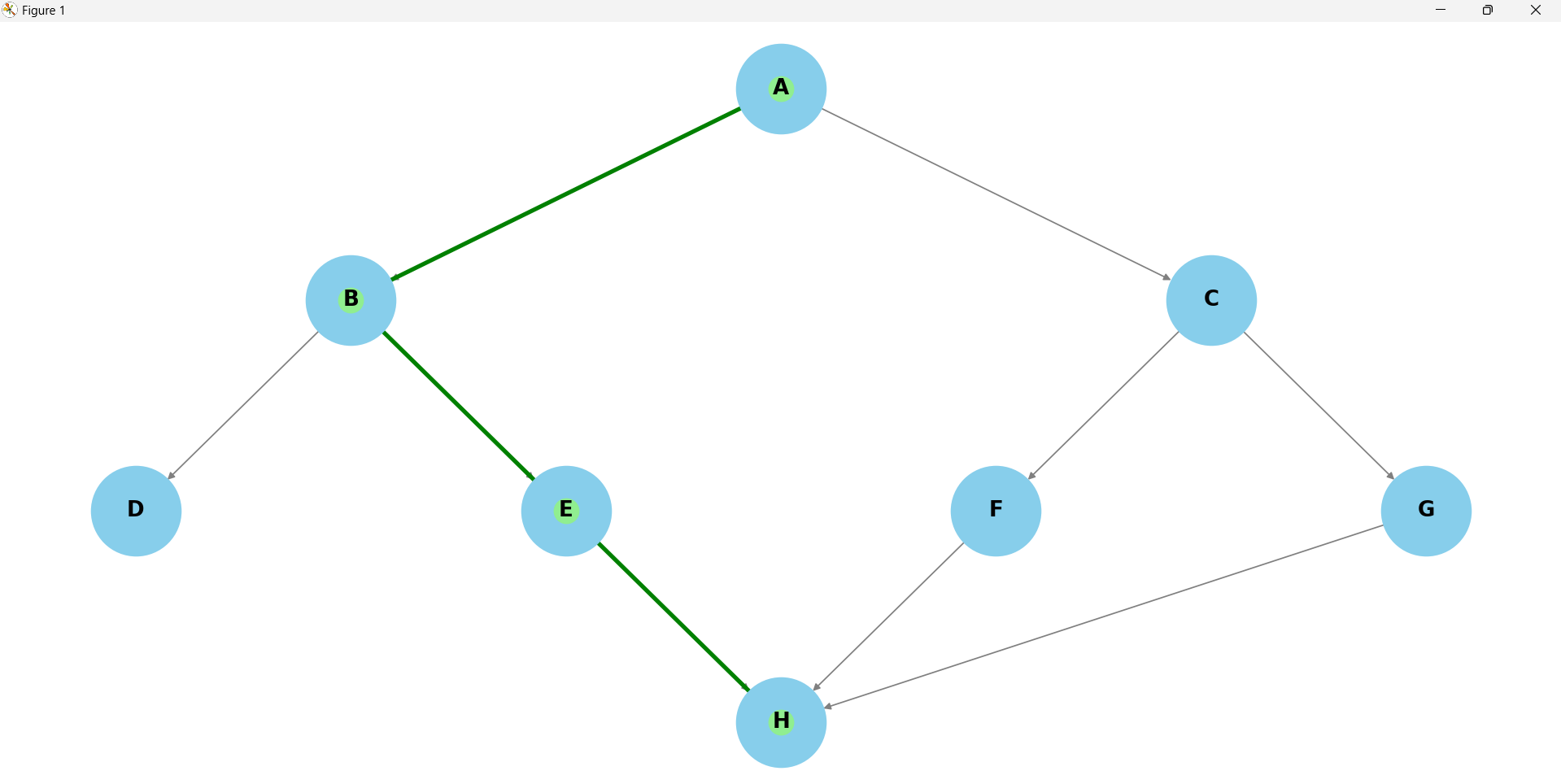
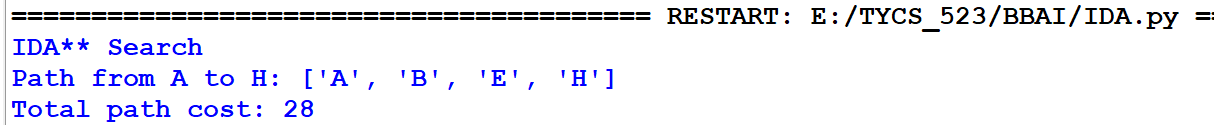
**else:**

**print("No path found.")**

# Visualize the graph and the path

**visualize\_graph(graph, result\_path, pos)**

**OUTPUT:**

****

1. **Beam Search**

**CODE:**

**import networkx as nx**

**import matplotlib.pyplot as plt**

**# Node class to store information about each node**

**class Node:**

**def \_\_init\_\_(self, name, g, h):**

**self.name = name**

**self.g = g # Cost from start to node**

**self.h = h # Heuristic to goal**

**self.f = g + h # f = g + h (total estimated cost)**

**def \_\_lt\_\_(self, other):**

**return self.f < other.f**

**# Beam Search for the graph**

**def beam\_search(graph, start, goal, heuristic, beam\_width):**

**# Initialize the frontier with the start node**

**frontier = [Node(start, 0, heuristic[start])]**

**explored = set() # To keep track of explored nodes**

**path = {start: None} # To track the path from start to goal**

**while frontier:**

**# Sort the frontier by heuristic value (best-first)**

**frontier.sort(key=lambda node: node.f)**

**# Only keep the top `beam\_width` nodes in the frontier**

**frontier = frontier[:beam\_width]**

**new\_frontier = []**

**for current\_node in frontier:**

**node = current\_node.name**

**if node == goal:**

**return reconstruct\_path(path, start, goal)**

**explored.add(node)**

**# Explore the neighbors of the current node**

**for neighbor in graph[node]:**

**if neighbor not in explored:**

**new\_frontier.append(Node(neighbor, current\_node.g + 1, heuristic[neighbor]))**

**if neighbor not in path:**

**path[neighbor] = node**

**# Add the newly explored frontier to the frontier list**

**frontier.extend(new\_frontier)**

**return None # If no path found**

**# Helper function to reconstruct the path from start to goal**

**def reconstruct\_path(path, start, goal):**

**current = goal**

**result\_path = []**

**while current is not None:**

**result\_path.append(current)**

**current = path[current]**

**result\_path.reverse()**

**return result\_path**

**# Function to visualize the graph and the path**

**def visualize\_graph(graph, path, pos):**

**G = nx.DiGraph() # Use a directed graph to show arrows**

**# Add edges to the graph**

**for node, neighbors in graph.items():**

**for neighbor in neighbors:**

**G.add\_edge(node, neighbor)**

**# Plot the graph**

**plt.figure(figsize=(10, 8))**

**# Draw the nodes and edges**

**nx.draw(**

**G, pos, with\_labels=True, node\_size=4000, node\_color='skyblue',**

**font\_size=15, font\_weight='bold', edge\_color='gray', arrows=True**

**)**

**# Highlight the path**

**if path:**

**path\_edges = list(zip(path, path[1:]))**

**nx.draw\_networkx\_edges(**

**G, pos, edgelist=path\_edges, edge\_color='green',**

**width=3, arrows=True, arrowstyle='-|>'**

**)**

**nx.draw\_networkx\_nodes(G, pos, nodelist=path, node\_color='lightgreen')**

**plt.title("Beam Search", size=20)**

**plt.show()**

**# Helper function to calculate the total path cost**

**def calculate\_path\_cost(path, heuristic):**

**total\_cost = 0**

**for node in path:**

**total\_cost += heuristic[node]**

**return total\_cost**

**# Graph definition**

**graph = {**

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

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

**'C': ['F', 'G'],**

**'E': ['H'],**

**'F': ['H'],**

**'G': ['H'],**

**'D': []**

**}**

**# Heuristic values**

**heuristic = {**

**'A': 13,**

**'B': 12,**

**'C': 4,**

**'D': 7,**

**'E': 3,**

**'F': 8,**

**'G': 2,**

**'H': 0**

**}**

**# Node positions for layout (Start 'A' at top)**

**pos = {**

**'A': (0, 3),**

**'B': (-1, 2),**

**'C': (1, 2),**

**'D': (-1.5, 1),**

**'E': (-0.5, 1),**

**'F': (0.5, 1),**

**'G': (1.5, 1),**

**'H': (0, 0)**

**}**

**# Perform Beam Search**

**start\_node = 'A'**

**goal\_node = 'H'**

**beam\_width = 2 # Adjust the beam width (number of nodes to explore at each step)**

**# Execute Beam Search for the path**

**result\_path = beam\_search(graph, start\_node, goal\_node, heuristic, beam\_width)**

**if result\_path:**

**print("Beam Search")**

**print(f"Path from {start\_node} to {goal\_node}: {result\_path}")**

**# Calculate and display the total path cost**

**path\_cost = calculate\_path\_cost(result\_path, heuristic)**

**print(f"Total path cost: {path\_cost}")**

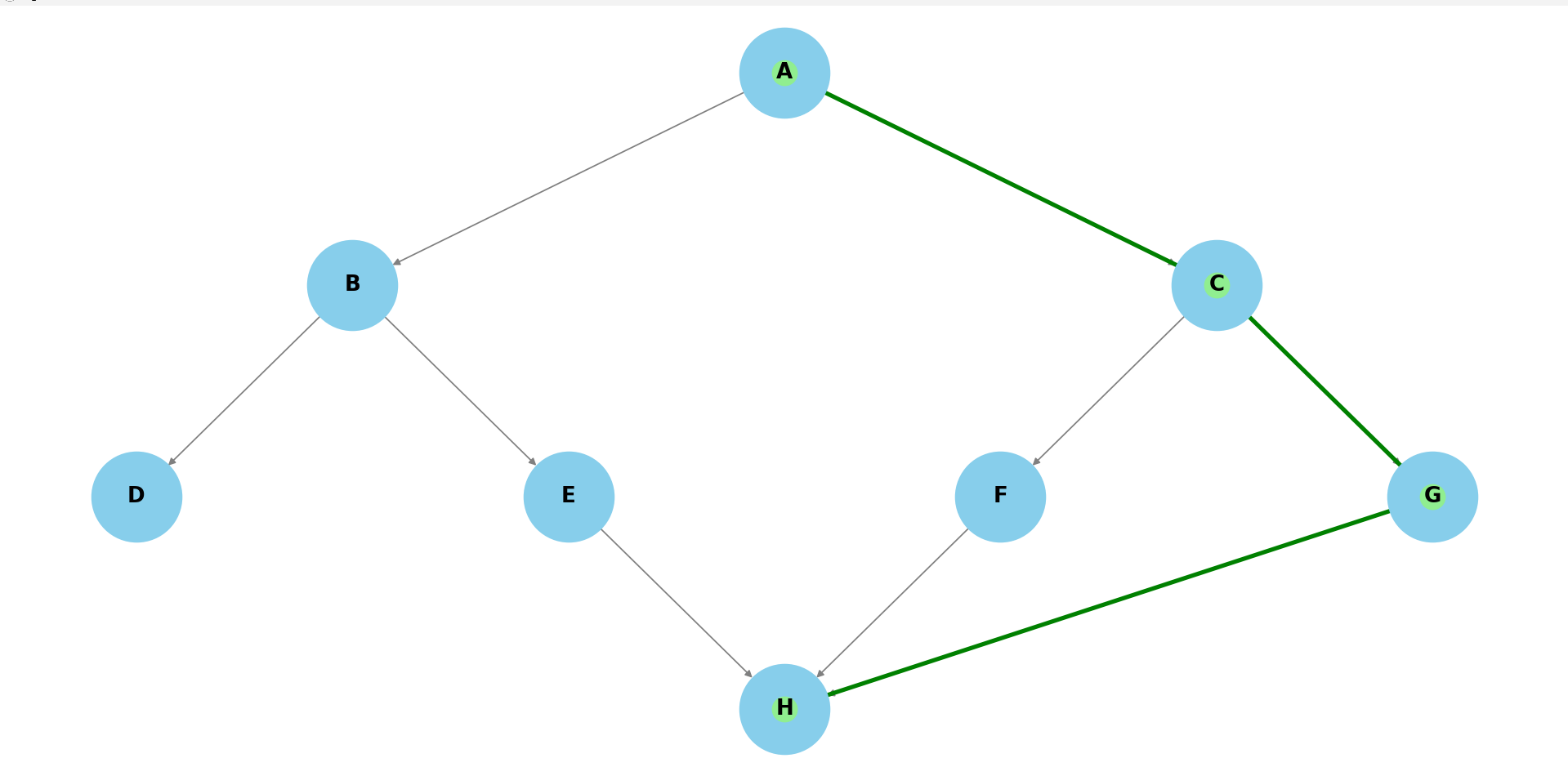
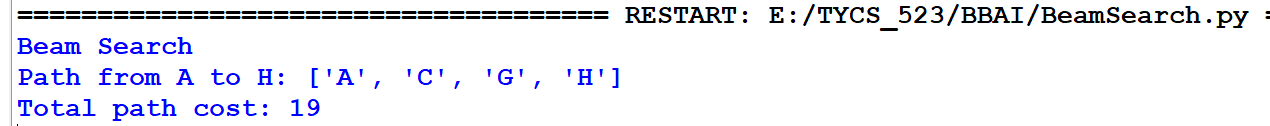
**else:**

**print("No path found.")**

**# Visualize the graph and the path**

**visualize\_graph(graph, result\_path, pos)**

**OUTPUT:**

****

**PRACTICAL NO 1**

**AIM: Implement Uninformed Search Algorithms:(Use Romania Map)**

1. **Breadth First Search**

**CODE:  
OUTPUT:**

**B. Depth First search**

**C. Depth Limited Search**

**D. Iterative Deepening Search**

**E. Uniform Cost Search**