**Tower of Hanoi**

print(“Tower Of Hanoi”)

def tower\_of\_hanoi(n, source, auxiliary, destination):

if n == 1:

print(f"Move disk 1 from {source} to {destination}")

return

tower\_of\_hanoi(n-1, source, auxiliary, destination)

print(f"Move disk {n} from {source} to {destination}")

tower\_of\_hanoi(n-1, auxiliary, source, destination)

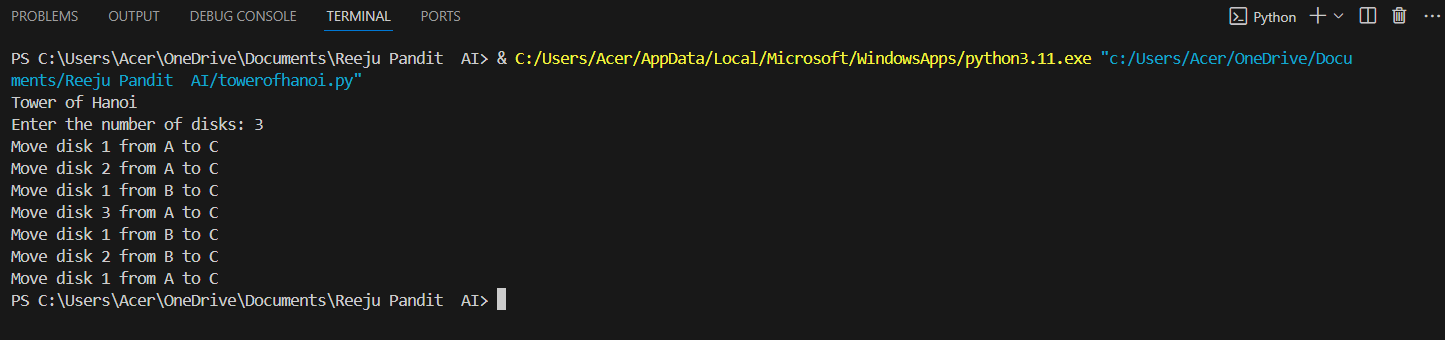
*# Ask the user for the number of disks*

num\_disks = int(input("Enter the number of disks: "))

*# Initial call to the function with source 'A', auxiliary 'B', and destination 'C'*

tower\_of\_hanoi(num\_disks, 'A', 'B', 'C')

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**

**Water Jug Problem**

print("Water Jug Problem")

*# Initial values for the jugs*

x = int(input("Enter the initial amount in Jug1 (x): "))

y = int(input("Enter the initial amount in Jug2 (y): "))

while True:

print("Rules:")

print("1. Fill Jug1 to its capacity (4 liters)")

print("2. Fill Jug2 to its capacity (3 liters)")

print("5. Empty Jug1")

print("6. Empty Jug2")

print("7. Pour water from Jug2 to Jug1 until Jug1 is full or Jug2 is empty")

print("8. Pour water from Jug1 to Jug2 until Jug2 is full or Jug1 is empty")

print("9. Pour all water from Jug2 into Jug1")

print("10. Pour all water from Jug1 into Jug2")

r = int(input("Enter the rule number (1-10): "))

if r == 1:

if x < 4: *# Fill Jug1 to its capacity (4 liters*)

x = 4

elif r == 2:

if y < 3: *# Fill Jug2 to its capacity (3 liters)*

y = 3

elif r == 5:

if x > 0: *# Empty Jug1*

x = 0

elif r == 6:

if y > 0: *# Empty Jug2*

y = 0

elif r == 7:

if x + y >= 4 and y > 0:

*# Pour water from Jug2 to Jug1 until Jug1 is full or Jug2 is empty*

x, y = min(4, x + y), max(0, y - (4 - x))

elif r == 8:

if x + y >= 3 and x > 0:

*# Pour water from Jug1 to Jug2 until Jug2 is full or Jug1 is empty*

x, y = max(0, x - (3 - y)), min(3, x + y)

elif r == 9:

if x + y <= 4 and y > 0*: # Pour all water from Jug2 into Jug1*

x, y = x + y, 0

elif r == 10:

if x + y <= 3 and x > 0: *# Pour all water from Jug1 into Jug2*

x, y = 0, x + y

else:

print("Invalid rule number. Please try again.")

*# Display the current state of both jugs*

print("x =", x)

print("y =", y)

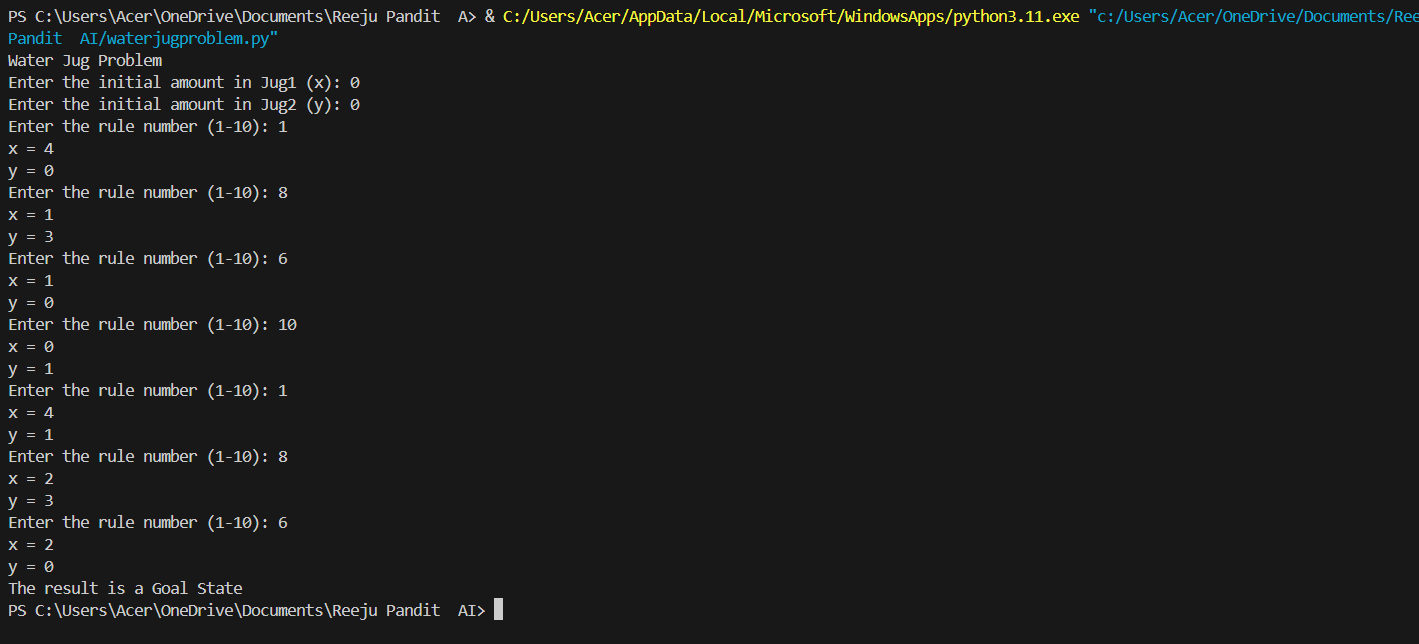
*# Check if the goal state (2 liters in Jug1) is reached*

if x == 2 and y == 0:

print("The result is a Goal State")

break

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**

**Simple ChatBot**

# Simple Chatbot about IOST, TU courses

responses = {

"hello": "Hi! Welcome to IOST, TU course chatbot. How can I help you?",

"about": "The Institute of Science and Technology (IOST) is a constituent campus of Tribhuvan University (TU) that offers various undergraduate and graduate programs in science and technology.",

"courses": "We offer the following courses: Bachelor of Science in Computer Science and Information Technology (BSc CSIT), Bachelor of Science in Electronics and Communication Engineering (BSc ECE), Bachelor of Science in Civil Engineering (BSc CE), and more.",

"bsc csit": "Our BSc CSIT program is a four-year undergraduate degree that covers computer science and information technology. It includes courses on programming, data structures, algorithms, computer networks, and more.",

"bsc ece": "Our BSc ECE program is a four-year undergraduate degree that covers electronics and communication engineering. It includes courses on electronics, communication systems, microprocessors, and more.",

"bsc ce": "Our BSc CE program is a four-year undergraduate degree that covers civil engineering. It includes courses on structural analysis, transportation engineering, water resources engineering, and more.",

"masters": "We also offer various master's programs, including Master of Science in Computer Science and Information Technology (MSc CSIT), Master of Science in Electronics and Communication Engineering (MSc ECE), and more.",

"eligibility": "The eligibility criteria for our undergraduate programs is a minimum of 45% in 10+2 or equivalent in science stream. For master's programs, a bachelor's degree in a relevant field is required.",

"fee": "The course fee for our undergraduate programs is NPR 1,20,000 per year, and for master's programs, it is NPR 1,50,000 per year.",

"bsc csit fee": "The course fee for our BSc CSIT program is NPR 1,20,000 per year.",

"bsc ece fee": "The course fee for our BSc ECE program is NPR 1,20,000 per year.",

"bsc ce fee": "The course fee for our BSc CE program is NPR 1,20,000 per year.",

"msc csit fee": "The course fee for our MSc CSIT program is NPR 1,50,000 per year.",

"msc ece fee": "The course fee for our MSc ECE program is NPR 1,50,000 per year.",

"query": "Please ask your query, and I'll do my best to answer it."

}

def chatbot():

print("Welcome to IOST, TU course chatbot!")

while True:

user\_input = input("You: ").lower()

if user\_input in responses:

print("Chatbot:", responses[user\_input])

elif user\_input == "exit":

print("Chatbot: Goodbye!")

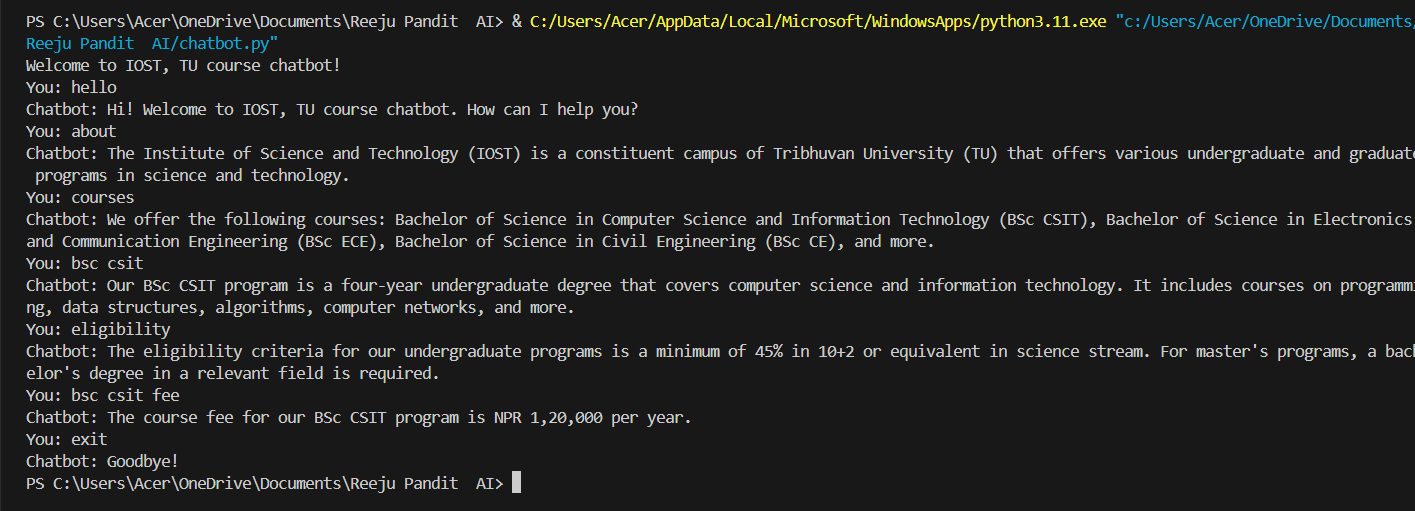
break

else:

print("Chatbot: Sorry, I didn't understand that. Please try again.")

chatbot()

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**

**Breadth Search Frist**

print(“Breadth First Search”)

from collections import deque

graph = { *# Define the graph*

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

'B': ['C'],

'C': [],

'D': [],

'E': ['F'],

'F': ['G', 'H'],

'G': [],

'H': [],

}

def bfs\_tree(graph, start):

visited = []

queue = deque([start])

parent = {start: None}

while queue:

node = queue.popleft()

if node not in visited:

visited.append(node)

print(node, end=" ")

for neighbour in graph[node]:

if neighbour not in visited:

queue.append(neighbour)

parent[neighbour] = node

return parent

def print\_tree(parent, root):

print("\nTree structure:")

for node in parent:

if parent[node] is not None:

print(f"{parent[node]} -> {node}")

def main():

root\_node = input("Enter the root node: ")

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

print("Following is the Breadth-First Search")

parent = bfs\_tree(graph, root\_node)

print\_tree(parent, root\_node)

*# Additional logic to find the path from root to goal*

path = []

current\_node = goal\_node

while current\_node is not None:

path.append(current\_node)

current\_node = parent.get(current\_node)

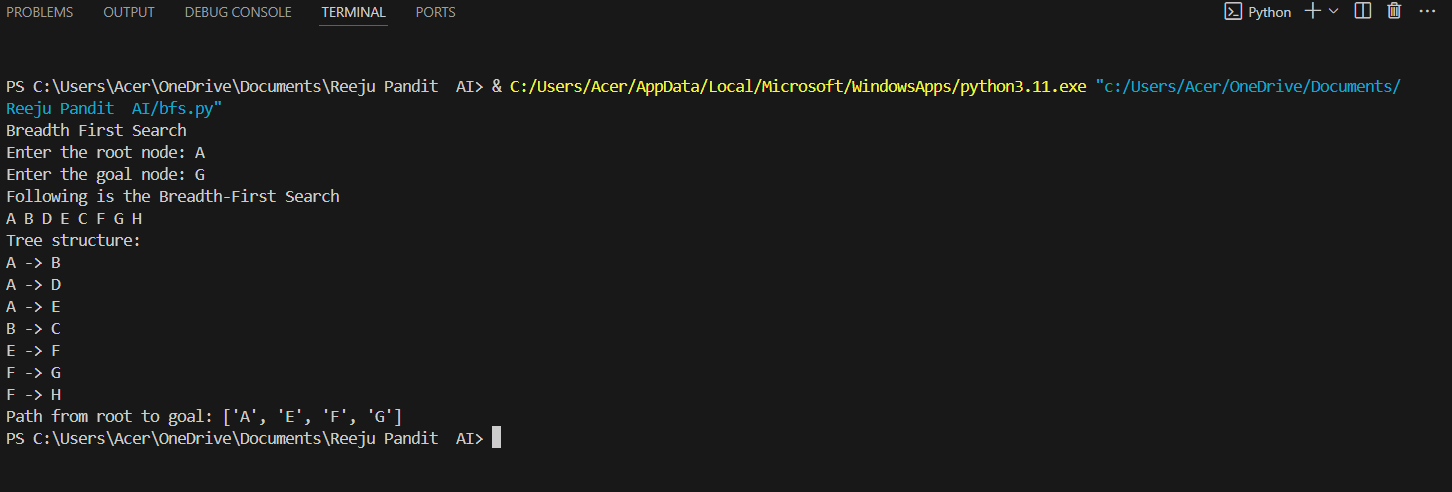
path.reverse()

print("Path from root to goal:", path)

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

main()

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**

**Best Search First:**

print(“Best First Search”)

from queue import PriorityQueue

v = 14

graph = [[] for \_ in range(v)]

def best\_first\_search(actual\_src, target, n): *# Function for implementing Best First Search*

visited = [False] \* n

pq = PriorityQueue()

pq.put((0, actual\_src))

visited[actual\_src] = True

while not pq.empty():

u = pq.get()[1]

print(u, end=" ")  *# Displaying the path having lowest cost*

if u == target:

break

for v, c in graph[u]:

if not visited[v]:

visited[v] = True

pq.put((c, v))

print()

def addedge(x, y, cost): *# Function for adding edges to the graph*

graph[x].append((y, cost))

graph[y].append((x, cost))

addedge(0, 1, 3) *# Adding edges to the graph*

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

addedge(9, 13, 2)

source = int(input("Enter the source node: ").strip()) *# Asking user for source and target nodes*

target = int(input("Enter the target node: ").strip())

if 0 <= source < v and 0 <= target < v: *# Ensure the source and target are within the valid range*

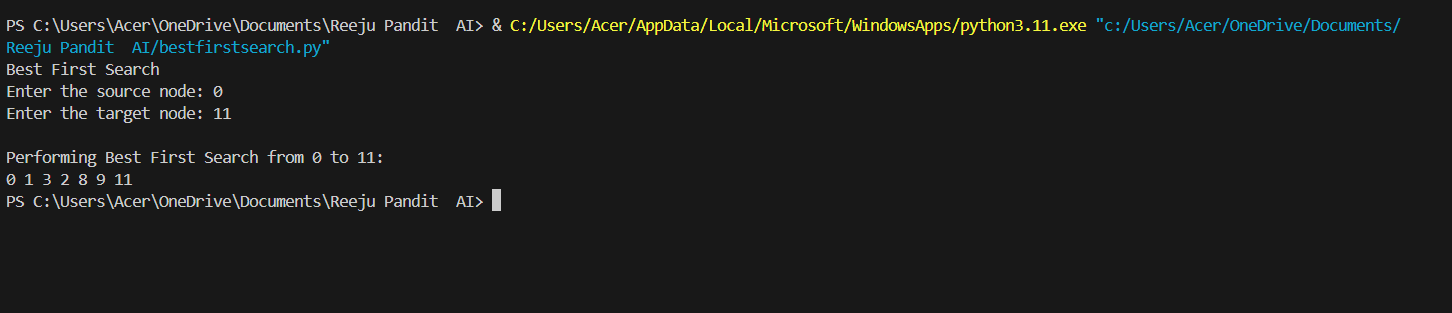
print(f"\nPerforming Best First Search from {source} to {target}:")

best\_first\_search(source, target, v)

else:

print("Invalid node values. Please enter values between 0 and", v-1)

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**

**Depth First Search:**

print(“Depth First Search”)

from collections import deque

*# Define the graph*

graph = {

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

'B': ['C'],

'C': [],

'D': [],

'E': ['F'],

'F': ['G', 'H'],

'G': [],

'H': [],

}

def dfs\_tree(graph, start):

visited = set()

stack = [start]

parent = {start: None}

while stack:

node = stack.pop()

if node not in visited:

visited.add(node)

print(node, end=" ")

for neighbour in graph[node]:

if neighbour not in visited:

stack.append(neighbour)

parent[neighbour] = node

return parent

def print\_tree(parent, root):

print("\nTree structure:")

for node in parent:

if parent[node] is not None:

print(f"{parent[node]} -> {node}")

def main():

root\_node = input("Enter the root node: ")

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

print("Following is the Depth-First Search")

parent = dfs\_tree(graph, root\_node)

print\_tree(parent, root\_node)

# Additional logic to find the path from root to goal

path = []

current\_node = goal\_node

while current\_node is not None:

path.append(current\_node)

current\_node = parent.get(current\_node)

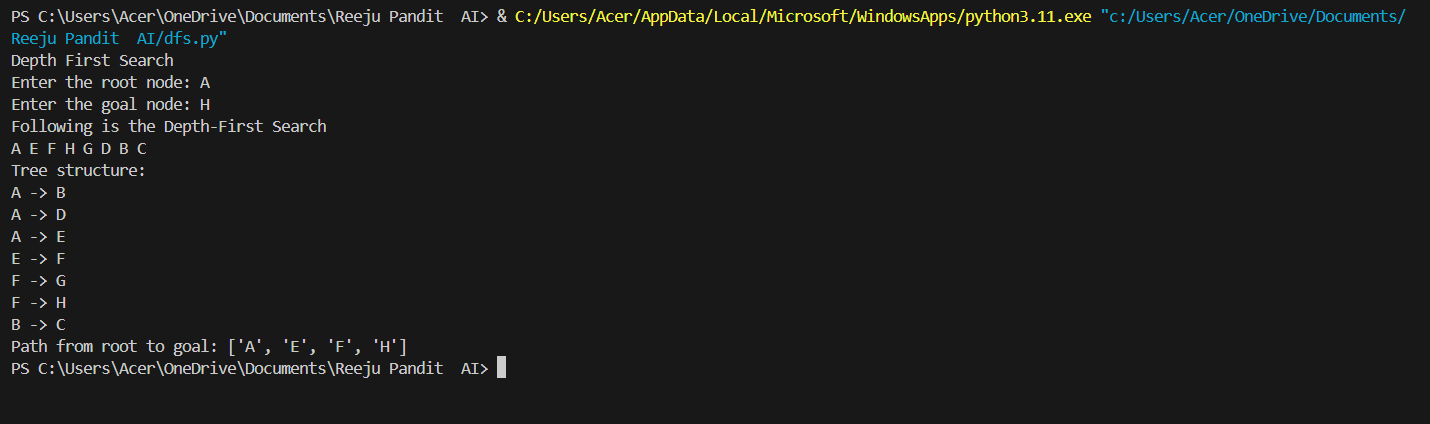
path.reverse()

print("Path from root to goal:", path)

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

main()

**Output:**

****

**A\* search:**

print(“A\* Search Algorithm ”)

import heapq

class Node:

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

self.name = name

self.heuristic = heuristic

self.g = float('inf')

self.f = float('inf')

self.parent = None

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

return self.f < other.f

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

start\_node = Node(start, heuristic(graph, start, goal))

goal\_node = Node(goal, 0)

open\_list = []

closed\_list = set()

heapq.heappush(open\_list, start\_node)

while open\_list:

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

closed\_list.add(current\_node.name)

if current\_node.name == goal:

path = []

while current\_node.parent:

path.append(current\_node.name)

current\_node = current\_node.parent

path.append(start)

path.reverse()

return path

for neighbor, cost in graph[current\_node.name].items():

if neighbor not in closed\_list:

neighbor\_node = Node(neighbor, heuristic(graph, neighbor, goal))

neighbor\_node.g = current\_node.g + cost

neighbor\_node.f = neighbor\_node.g + neighbor\_node.heuristic

neighbor\_node.parent = current\_node

heapq.heappush(open\_list, neighbor\_node)

return None

def heuristic(graph, node, goal):

*# Manhattan distance heuristic*

return abs(ord(node) - ord(goal))

*# Example graph*

graph = {

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

'B': {'A': 1, 'D': 2, 'E': 5},

'C': {'A': 4, 'F': 3},

'D': {'B': 2},

'E': {'B': 5, 'F': 1},

'F': {'C': 3, 'E': 1}

}

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

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

if start\_node in graph and goal\_node in graph:

path = a\_star\_search(graph, start\_node, goal\_node)

if path:

print("Path:", " -> ".join(path))

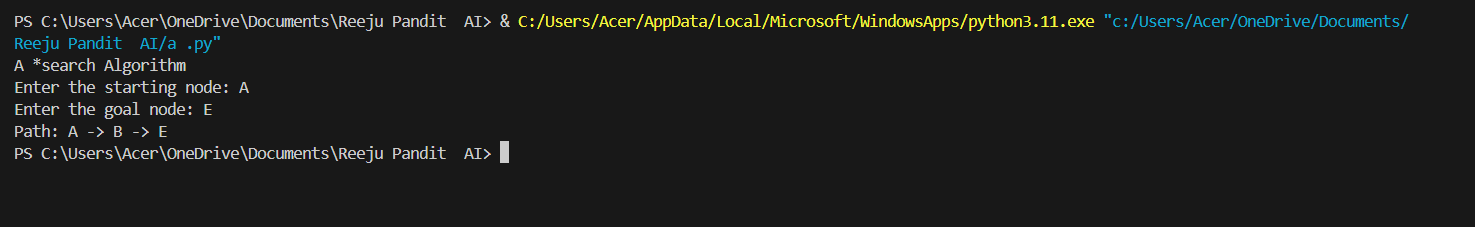
else:

print("No path found")

else:

print("Invalid node. Please enter a node that exists in the graph")

**Output:**

****

**Compiler: Visual Studio Code**

**Language: Python**