

**Department of**

**Artificial Intelligence and Data Science**

***B. Tech. Computer Science and Engineering***

***(Artificial Intelligence)***

**AIL333 - AI ALGORITHMS LAB**



***<STUDENT NAME>***

***<STUDENT ROLL NO>***

**LAB RECORD**

**November 2025**



**TABLE OF CONTENTS**

| **No.** | **Date** | **Experiments** | **Signature** |
| --- | --- | --- | --- |
| 1 |  | Echo Hello World |  |
| 2 |  | Identify Data Type |  |
| 3 |  | Arithmetic Operations |  |
| 4 |  | Boolean Operations |  |
| 5 |  | String Operations |  |
| 6 |  | If-Else - Check 3 Digit Number |  |
| 7 |  | Python Familiarity Using if-elif-else |  |
| 8 |  | Range() Function |  |
| 9 |  | For Loop |  |
| 10 |  | While Loop - Sum of Digits |  |
| 11 |  | Nested Loop - Prime Numbers |  |
| 12 |  | List Operations |  |
| 13 |  | Indexing and Slicing on a List |  |
| 14 |  | List Comprehension |  |
| 15 |  | Dictionary Operations |  |
| 16 |  | Tuple Operations |  |
| 17 |  | Set Operations |  |
| 18 |  | Functions Twice |  |
| 19 |  | Prime Number Check |  |
| 20 |  | Class and Objects - Encapsulation |  |
| 21 |  | Basic DFS Traversal |  |
| 22 |  | Subset Sum using DFS |  |
| 23 |  | Basic BFS Traversal |  |
| 24 |  | Subset Sum using BFS |  |
| 25 |  | Uniform Cost Search (UCS) |  |
| 26 |  | Depth Limited Search (DLS) |  |
| 27 |  | Greedy Best First Search |  |
| 28 |  | A\* Algorithm |  |
| 29 |  | CSP - N Queens - Backtracking |  |
| 30 |  | N-Queens using Hill Climbing |  |
| 31 |  | Propositional Logic Evaluation using SymPy |  |
| 32 |  | Propositional Logic - Validate Logical Rule |  |
| 33 |  | Brute Force - Travelling Salesman Problem (TSP) |  |

**Date:**

**Experiment 1: Echo Hello World**

**Aim**:  
To write a Python program that prints "Hello World".

**Algorithm**:

1. Start.
2. Use the print() function to display the message "Hello World".
3. End.

**Program in Python**:

print("Hello World")

**Output:**

Hello World

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 2:** Identify Data Type

**Aim**:  
To write a Python program that identifies the data type of given variables.

**Algorithm**:

1. Start.
2. Declare variables with different data types (int, float, string, boolean, list, tuple, dictionary).
3. Use the built-in function type() to check their types.
4. Print the results.
5. End.

**Program in Python**:

a = 10

b = 3.14

c = "AI Lab"

d = True

e = [1, 2, 3]

f = (4, 5, 6)

g = {"key": "value"}

print("Type of a:", type(a))

print("Type of b:", type(b))

print("Type of c:", type(c))

print("Type of d:", type(d))

print("Type of e:", type(e))

print("Type of f:", type(f))

print("Type of g:", type(g))

**Output:**

Type of a: <class 'int'>

Type of b: <class 'float'>

Type of c: <class 'str'>

Type of d: <class 'bool'>

Type of e: <class 'list'>

Type of f: <class 'tuple'>

Type of g: <class 'dict'>

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 3:** Arithmetic Operations

**Aim**:  
To perform arithmetic operations (addition, subtraction, multiplication, division, modulus, exponent, floor division) in Python

**Algorithm**:

1. Start.
2. Input two numbers.
3. Perform all arithmetic operations using operators: +, -, \*, /, %, \*\*, //.
4. Display the results.
5. End.

**Program in Python**:

a = 15

b = 4

print("Addition:", a + b)

print("Subtraction:", a - b)

print("Multiplication:", a \* b)

print("Division:", a / b)

print("Modulus:", a % b)

print("Exponent:", a \*\* b)

print("Floor Division:", a // b)

**Output:**

Addition: 19

Subtraction: 11

Multiplication: 60

Division: 3.75

Modulus: 3

Exponent: 50625

Floor Division: 3

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 4:** Boolean Operations

**Aim**:  
To perform boolean operations (AND, OR, NOT) in Python.

**Algorithm**:

1. Start the program.
2. Initialize boolean variables.
3. Apply logical operators (and, or, not).
4. Print results.
5. Stop.

**Program in Python**:

a = True

b = False

print("a AND b:", a and b)

print("a OR b:", a or b)

print("NOT a:", not a)

**Output:**

a AND b: False

a OR b: True

NOT a: False

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 5:** String Operations

**Aim**:  
To demonstrate string operations in Python.

**Algorithm**:

1. Start the program.
2. Define a string.
3. Perform concatenation, repetition, slicing, and length operations.
4. Print results.
5. Stop.

**Program in Python**:

s = "Python"

print("Concatenation:", s + " Programming")

print("Repetition:", s \* 3)

print("Slicing:", s[0:4])

print("Length:", len(s))

**Output:**

Concatenation: Python Programming

Repetition: PythonPythonPython

Slicing: Pyth

Length: 6

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 6:** If-Else – Check 3 Digit Number

**Aim**:  
To check whether a number is a 3-digit number using if-else.

**Algorithm**:

1. Start the program.
2. Input a number.
3. If the number is between 100 and 999 (inclusive), print "3-digit number".
4. Else, print "Not a 3-digit number".
5. Stop.

**Program in Python**:

n = int(input("Enter a number: "))

if 100 <= n <= 999:

print("3-digit number")

else:

print("Not a 3-digit number")

**Output:**

Enter a number: 200

3-digit number

Enter a number: 89

Not a 3-digit number

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 7:** Python Familiarity Using if-elif-else

**Aim**:  
To demonstrate decision-making using if-elif-else in Python.

**Algorithm**:

1. Start the program.
2. Input marks.
3. Use if-elif-else to classify grades.
4. Print grade.
5. Stop.

**Program in Python**:

marks = int(input("Enter marks: "))

if marks >= 90:

print("Grade: A")

elif marks >= 75:

print("Grade: B")

elif marks >= 50:

print("Grade: C")

else:

print("Grade: F")

**Output:**

Enter marks: 92

Grade: A

Enter marks: 67

Grade: C

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 8:** Range() Function

**Aim**:  
To demonstrate the use of the range() function in Python.

**Algorithm**:

1. Start the program.
2. Use range() in different ways: only stop, start-stop, start-stop-step.
3. Print the results.
4. Stop.

**Program in Python**:

print("range(5):", list(range(5)))

print("range(2, 10):", list(range(2, 10)))

print("range(1, 10, 2):", list(range(1, 10, 2)))

**Output:**

range(5): [0, 1, 2, 3, 4]

range(2, 10): [2, 3, 4, 5, 6, 7, 8, 9]

range(1, 10, 2): [1, 3, 5, 7, 9]

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 9:** For Loop

**Aim**:  
To demonstrate the use of a for loop in Python.

**Algorithm**:

1. Start the program.
2. Get start and end index
3. Use a for loop to print numbers from start to end.
4. Stop.

**Program in Python**:

s=int(input())

t=int(input())

for i in range(s,t):

print(i,end=" ")

print()

**Input:**

0

10

**Output**

0 1 2 3 4 5 6 7 8 9

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 10:** While Loop – Sum of Digits

**Aim**:  
To find the sum of digits of a number using a while loop.

**Algorithm**:

1. Start the program.
2. Input a number n.
3. Initialize sum = 0.
4. While n > 0:
   1. Extract digit = n % 10.
   2. Add digit to sum.
   3. Update n = n // 10.
5. Print sum.
6. Stop.

**Program in Python**:

n = int(input("Enter a number: "))

sum\_digits = 0

while n > 0:

digit = n % 10

sum\_digits += digit

n //= 10

print("Sum of digits:", sum\_digits)

**Input:**

1234

**Output:**

Sum of digits: 10

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 11:** Nested Loop – Prime Numbers

**Aim**:  
To generate prime numbers up to n using nested loops.

**Algorithm**:

1. Start the program.
2. Input a limit n.
3. For each number i from 2 to n:
   * Assume prime = True.
   * For each j from 2 to i-1:
     1. If i % j == 0, set prime = False and break.
   * If prime is True, print i.
4. Stop.

**Program in Python**:

n = int(input("Enter limit: "))

for i in range(2, n + 1):

prime = True

for j in range(2, i):

if i % j == 0:

prime = False

break

if prime:

print(i, end=" ")

**Input:**

10

**Output:**

2 3 5 7

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 12:** List Operations

**Aim**:  
To perform basic list operations in Python.

**Algorithm**:

1. Start the program.
2. Create a list with some numbers.
3. Demonstrate append, insert, remove, pop, sort, reverse operations.
4. Print results.
5. Stop

**Program in Python**:

numbers = [10, 20, 30]

numbers.append(40)

numbers.insert(1, 15)

numbers.remove(30)

popped = numbers.pop()

numbers.sort()

numbers.reverse()

print("Final list:", numbers)

print("Popped element:", popped)

**Output:**

Final list: [20, 15, 10]

Popped element: 40

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 13:** Indexing and Slicing on a List

**Aim**:  
To demonstrate indexing and slicing operations on lists in Python.

**Algorithm**:

1. Start the program.
2. Create a list with numbers.
3. Access elements using positive and negative indices.
4. Slice the list in different ranges.
5. Print results.
6. Stop.

**Program in Python**:

nums = [10, 20, 30, 40, 50]

print("First element:", nums[0])

print("Last element:", nums[-1])

print("Slice [1:4]:", nums[1:4])

print("Slice [:3]:", nums[:3])

print("Slice [2:]:", nums[2:])

**Output:**

First element: 10

Last element: 50

Slice [1:4]: [20, 30, 40]

Slice [:3]: [10, 20, 30]

Slice [2:]: [30, 40, 50]

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 14:** List Comprehension

**Aim**:  
To demonstrate list comprehension in Python.

**Algorithm**:

1. Start the program.
2. Create a list of numbers using list comprehension.
3. Apply a condition inside comprehension.
4. Print results.
5. Stop.

**Program in Python**:

squares = [x\*\*2 for x in range(1, 6)]

evens = [x for x in range(10) if x % 2 == 0]

print("Squares:", squares)

print("Even numbers:", evens)

**Output:**

Squares: [1, 4, 9, 16, 25]

Evens: [0, 2, 4, 6, 8]

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 15:** Dictionary Operations

**Aim**:  
To perform dictionary operations in Python.

**Algorithm**:

1. Start the program.
2. Create a dictionary with key-value pairs.
3. Demonstrate add, update, delete, access, and traversal operations.
4. Print results.
5. Stop.

**Program in Python**:

books = {"Python Basics": 10, "Data Science": 5, "AI Intro": 7}

while True:

print("\n1. Add Book\n2. Update Stock\n3. Delete Book\n4. Display\n5. Exit")

choice = int(input("Enter choice: "))

if choice == 1:

title = input("Enter book title: ")

qty = int(input("Enter quantity: "))

books[title] = qty

elif choice == 2:

title = input("Enter book title to update: ")

if title in books:

qty = int(input("Enter new quantity: "))

books[title] = qty

else:

print("Book not found.")

elif choice == 3:

title = input("Enter book title to delete: ")

if title in books:

del books[title]

else:

print("Book not found.")

elif choice == 4:

print("Books in stock:")

for title, stock in books.items():

print(f"{title}: {stock}")

elif choice == 5:

break

else:

print("Invalid choice.")

**Output:**

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 1

Enter book title: Python

Enter quantity: 3

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 1

Enter book title: Java

Enter quantity: 5

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 2

Enter book title to update: Python

Enter new quantity: 2

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 4

Books in stock:

Python Basics: 10

Data Science: 5

AI Intro: 7

Python: 2

Java: 5

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 3

Enter book title to delete: Java

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 4

Books in stock:

Python Basics: 10

Data Science: 5

AI Intro: 7

Python: 2

1. Add Book

2. Update Stock

3. Delete Book

4. Display

5. Exit

Enter choice: 5

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 16: Tuple Operations**

**Aim**:  
To demonstrate tuple operations in Python.

**Algorithm**:

1. Start the program.
2. Create a tuple.
3. Access elements using indexing and slicing.
4. Use functions like count() and index().
5. Print results.
6. Stop.

**Program in Python**:

t = (10, 20, 30, 20, 40)

print("Tuple:", t)

print("First element:", t[0])

print("Slice [1:4]:", t[1:4])

print("Count of 20:", t.count(20))

print("Index of 30:", t.index(30))

**Output:**

Tuple: (10, 20, 30, 20, 40)

First element: 10

Slice [1:4]: (20, 30, 20)

Count of 20: 2

Index of 30: 2

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 17: Set Operations**

**Aim**:  
To perform set operations in Python.

**Algorithm**:

1. Start the program.
2. Create two sets.
3. Perform union, intersection, difference.
4. Add and remove elements.
5. Print results.
6. Stop.

**Program in Python**:

A = {1, 2, 3, 4}

B = {3, 4, 5, 6}

print("Union:", A | B)

print("Intersection:", A & B)

print("Difference A-B:", A - B)

A.add(7)

A.remove(2)

print("Updated Set A:", A)

**Output:**

Union: {1, 2, 3, 4, 5, 6}

Intersection: {3, 4}

Difference A-B: {1}

Updated Set A: {1, 3, 4, 7}

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 18: Functions Twice**

**Aim**:  
To demonstrate user-defined functions by writing a function that prints a message twice.

**Algorithm**:

1. Start the program.
2. Define a function print\_twice(msg).
3. Inside the function, print the message two times.
4. Call the function with a string.
5. Stop.

**Program in Python**:

def print\_twice(msg):

print(msg)

print(msg)

print\_twice("Hello Functions")

**Output:**

Hello Functions

Hello Functions

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 19: Prime Number Check**

**Aim**:  
To check whether a given number is prime.

**Algorithm**:

1. Start the program.
2. Input a number n.
3. If n <= 1, it is not prime.
4. Check divisibility from 2 to n-1.
5. If divisible, not prime; else, prime.
6. Stop.

**Program in Python**:

n = int(input("Enter a number: "))

if n <= 1:

print("Not Prime")

else:

for i in range(2, n):

if n % i == 0:

print("Not Prime")

break

else:

print("Prime Number")

**Output:**

Enter a number: 7

Prime Number

Enter a number: 9

Not Prime

**Result**:

The program is successfully executed.

**Date:**

**Experiment 20: Class and Objects – Encapsulation**

**Aim**:  
To demonstrate encapsulation using classes and objects in Python.

**Algorithm**:

1. Start the program.
2. Define a class Student with private attributes.
3. Provide setter and getter methods.
4. Create an object and set values.
5. Access values using getters.
6. Stop.

**Program in Python**:

class Student:

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

self.\_\_name = ""

def set\_name(self, name):

self.\_\_name = name

def get\_name(self):

return self.\_\_name

s = Student()

s.set\_name("Alice")

print("Student Name:", s.get\_name())

**Output:**

Student Name: Alice

**Result**:  
The program is successfully executed.

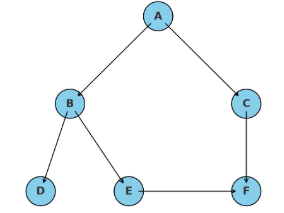
**Date:**

**Experiment 21: Basic DFS Traversal**

**Aim**:  
To implement Depth First Search (DFS) on a graph

**Algorithm**:

1. Start
2. Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
3. Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
4. Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.
5. Stop

**Program in Python**:

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

visited = set()

def dfs(node):

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbour in graph[node]:

dfs(neighbour)

dfs('A')

**Output:**

A B D E F C

**Result**:  
The program is successfully executed.

**Date:**

### **Experiment 22:** Subset Sum using DFS

**Aim**:  
To solve the subset sum problem using DFS.

**Algorithm**:

1. Initialize structures
   1. Create an empty set visited to track visited nodes.
   2. Set a global flag found = False to stop search once the goal is reached.
2. Define DFS(node, goal):
   1. If node is not visited and found == False:
      1. Print the current node (traversal step).
      2. Compute the sum of the subset assigned to this node (solutions[node]).
      3. If this sum equals goal:
         1. Set found = True.
         2. Print "Goal Reached" and display the solution subset.
      4. Add the current node to visited.
      5. For each neighbour in graph[node]:
         1. Recursively call DFS(neighbour, goal).
3. Driver Code:
   1. Print "Following is the Depth-First Search".
   2. Call DFS('a', goal) starting from node 'a'.
4. Termination:
   1. DFS stops when either:
      1. All reachable nodes are visited, or
      2. A subset matching the goal is found (found = True).

**Program in Python**:

S = {1,2,3}

Goal = 4

import numpy as np

graph = {

'a' : ['b','c','d'],

'b' : ['e','f','a'],

'c' : ['e','g','a'],

'd' : ['f','g','a'],

'e' : ['h','b','c'],

'f' : ['h','b','d'],

'g' : ['h','d','c'],

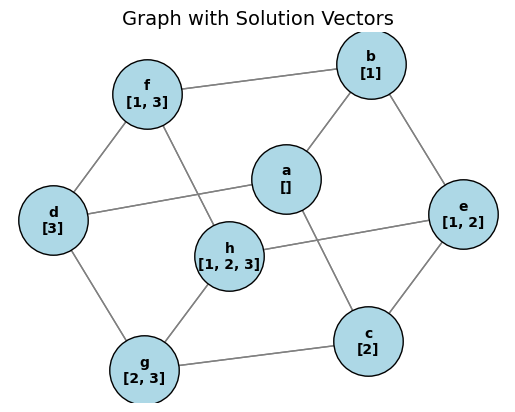
'h' : ['e','f','g']

}

solutions = {

'a' : [],

'b' : [1],

'c' : [2],

'd' : [3],

'e' : [1,2],

'f' : [1,3],

'g' : [2,3],

'h' : [1,2,3]

}

goal = 4

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

found = False

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

global found

if node not in visited and not found:

print (node)

if np.sum(solutions[node]) == goal:

found = True

print('Goal Reached')

print('Solution :' + str(solutions[node]))

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour, goal)

# Driver Code

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

dfs(visited, graph, 'a', 4)

**Output:**

Following is the Depth-First Search

a

b

e

h

f

Goal Reached

Solution :[1, 3]

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 23:Basic BFS Traversal**

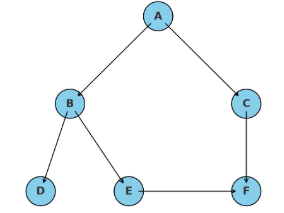
**Aim**:  
To implement Breadth First Search (BFS) on a graph.

**Algorithm**:

1. Start the program.
2. Represent the graph using adjacency list.
3. Initialize a queue with the starting node.
4. While queue not empty:  
   1. Pop first node.
   2. If not visited, print it and enqueue its neighbors.
5. Stop.

**Program in Python**:

from collections import deque



graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

def bfs(start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(graph[node])

bfs('A')

**Output:** A B C D E F

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 24: Subset Sum using BFS**

**Aim**:  
To solve the subset sum problem using BFS.

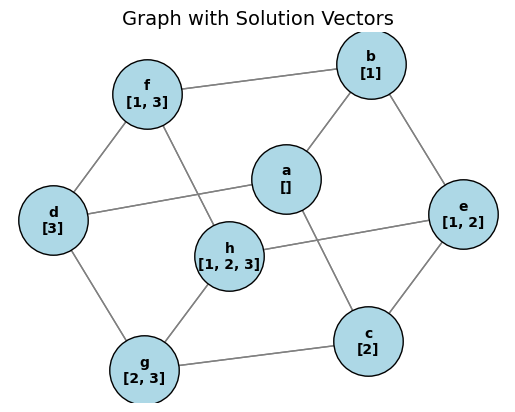
**Algorithm**:

1. Initialize:
   1. Create an empty list visited to track visited nodes.
   2. Create an empty list queue to manage BFS traversal.
   3. Set a flag found = False to indicate if the goal has been reached.
2. Start BFS:
   1. Add the starting node to visited and queue.
3. While queue is not empty and goal not found:
   1. Remove the first node m from the queue.
   2. Print the current node m (for traversal trace).
   3. Check if the sum of values in solutions[m] equals the goal:
      1. If yes:
         1. Set found = True
         2. Print "Goal Reached"
         3. Print the solution list
         4. Exit the loop
   4. For each neighbor of node m in graph[m]:
      1. If neighbor is not in visited:
         1. Add neighbor to visited
         2. Add neighbor to queue
4. End BFS:
   1. If the goal is found, the search stops early.
   2. Otherwise, it continues until all reachable nodes are visited.

**Program in Python**:

import numpy as np

graph = {

'a' : ['b','c','d'],

'b' : ['e','f','a'],

'c' : ['e','g','a'],

'd' : ['f','g','a'],

'e' : ['h','b','c'],

'f' : ['h','b','d'],

'g' : ['h','d','c'],

'h' : ['e','f','g']

}

solutions = {

'a' : [],

'b' : [1],

'c' : [2],

'd' : [3],

'e' : [1,2],

'f' : [1,3],

'g' : [2,3],

'h' : [1,2,3]

}

goal = 4

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

found = False

def bfs(visited, graph, node,goal): #function for BFS

global found

visited.append(node)

queue.append(node)

while queue and not found: # Creating loop to visit each node

m = queue.pop(0)

print (m)

if np.sum(solutions[m]) == goal:

found = True

print('Goal Reached')

print('Solution :' + str(solutions[m]))

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

# Driver Code

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

bfs(visited, graph, 'a',4) # function calling

**Output:**

Following is the Breadth-First Search

a

b

c

d

e

f

Goal Reached

Solution :[1, 3]

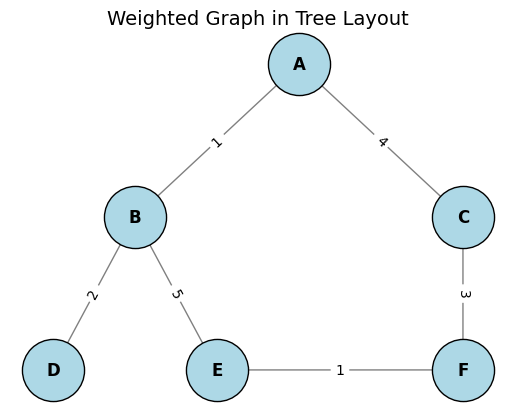
**Result**:  
The program is successfully executed.

**Date:**

**Experiment 25: Uniform Cost Search (UCS)**

**Aim**:  
To implement Uniform Cost Search (UCS) for finding the least-cost path.

**Algorithm**:

1. Start the program.
2. Use priority queue (min-heap) with (cost, node, path).
3. Always expand the node with least cost.
4. If goal found, print cost and path.
5. Stop.

**Program in Python**:

import heapq

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('D', 2), ('E', 5)],

'C': [('F', 3)],

'D': [],

'E': [('F', 1)],

'F': []

}

def ucs(start, goal):

visited = set()

pq = [(0, start, [start])] # cost, node, path

while pq:

cost, node, path = heapq.heappop(pq)

if node == goal:

print("Path:", path, "Cost:", cost)

return

if node not in visited:

visited.add(node)

for neighbor, edge\_cost in graph[node]:

heapq.heappush(pq, (cost + edge\_cost, neighbor, path + [neighbor]))

ucs('A', 'F')

**Output:**

Path: ['A', 'B', 'E', 'F'] Cost: 7

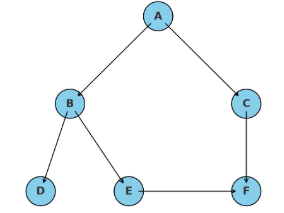
**Result**:  
The program is successfully executed.

**Date:**

**Experiment 26: Depth Limited Search (DLS)**

**Aim**:  
To implement Depth Limited Search in Python.

**Algorithm**:

1. Start the program.
2. Define recursive DFS with depth limit.
3. If node == goal, return success.
4. If depth limit reached, stop recursion.
5. Else, expand children.
6. Stop.

**Program in Python**:

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

def dls(node, goal, limit, depth=0):

if node == goal:

return True

if depth == limit:

return False

for neighbor in graph[node]:

if dls(neighbor, goal, limit, depth + 1):

return True

return False

print("A - F: Depth Limit 1 - Found:", dls('A', 'F', 1)) # limit 1

print("A - F: Depth Limit 2 - Found:", dls('A', 'F', 2)) # limit 2

**Output:**

A - F: Depth Limit 1 - Found: False

A - F: Depth Limit 2 - Found: True

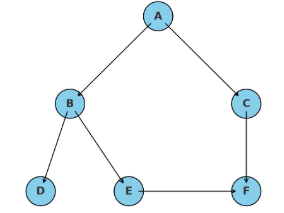
**Result**:  
The program is successfully executed.

**Date:**

**Experiment 27: Greedy Best First Search (GBFS)**

**Aim**:  
To implement Greedy Best First Search for finding a path using heuristic values.

**Algorithm**:

1. Start the program.
2. Use a priority queue with (heuristic, node, path).
3. Always expand the node with the lowest heuristic value.
4. If goal is reached → print path.
5. Stop.

**Program in Python**:

import heapq

graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

heuristic = {'A': 6, 'B': 4, 'C': 2, 'D': 7, 'E': 1, 'F': 0}

def greedy\_bfs(start, goal):

pq = [(heuristic[start], start, [start])]

visited = set()

while pq:

h, node, path = heapq.heappop(pq)

if node == goal:

print("Path:", path)

return

if node not in visited:

visited.add(node)

for neighbor in graph[node]:

heapq.heappush(pq, (heuristic[neighbor], neighbor, path + [neighbor]))

greedy\_bfs('A', 'F')

**Output:** Path: ['A', 'C', 'F']

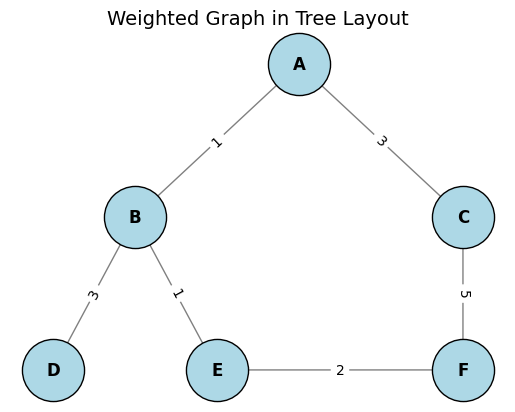
**Result**:  
The program is successfully executed.

**Date:**

**Experiment 28: A\* Algorithm**

**Aim**:  
To implement A\* algorithm for finding the optimal path

**Algorithm**:

1. Start the program.
2. Maintain an open list (priority queue) with (f = g + h, node, path, cost).
3. Expand the node with lowest f.
4. If node == goal → return path and cost.
5. Else expand neighbors and push (new\_g + h, neighbor).
6. Stop.

**Program in Python**:

import heapq

graph = {

'A': [('B', 1), ('C', 3)],

'B': [('D', 3), ('E', 1)],

'C': [('F', 5)],

'D': [],

'E': [('F', 2)],

'F': []

}

heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 6, 'E': 1, 'F': 0}

def a\_star(start, goal):

pq = [(heuristic[start], 0, start, [start])] # f, g, node, path

visited = set()

while pq:

f, g, node, path = heapq.heappop(pq)

if node == goal:

print("Path:", path, "Cost:", g)

return

if node not in visited:

visited.add(node)

for neighbor, cost in graph[node]:

new\_g = g + cost

new\_f = new\_g + heuristic[neighbor]

heapq.heappush(pq, (new\_f, new\_g, neighbor, path + [neighbor]))

a\_star('A', 'F')

**Output:**

Path: ['A', 'B', 'E', 'F'] Cost: 4

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 29: CSP – N Queens (Backtracking)**

**Aim**:  
To solve the N-Queens problem using backtracking.

**Algorithm**:

1. **Start**
2. Create an empty N x N chessboard initialized with 0.  
   1. 0 → empty cell
   2. 1 → cell occupied by a queen
3. Define a function is\_safe(board, row, col, N) that checks:  
   1. No queen exists in the **same row** to the left.
   2. No queen exists in the **upper-left diagonal**.
   3. No queen exists in the **lower-left diagonal**.
   4. If all checks pass, return **True**, otherwise **False**.
4. Define a recursive function solve(board, col, N) that tries to place queens column by column:  
   1. If col >= N, all queens are placed → **print solution**.
   2. For each row i in the current column col:  
      1. If is\_safe(board, i, col, N) is true:  
         1. Place queen at (i, col) (set board[i][col] = 1).
         2. Recursively call solve(board, col + 1, N).
         3. If solution found, continue; else **backtrack** by setting board[i][col] = 0.
5. Call solve(board, 0, N) starting from column 0.
6. If no configuration works, print **“No solution exists”**.
7. **End**

**Program in Python**:

def print\_solution(board, N):

"""Helper function to print the board."""

for i in range(N):

for j in range(N):

print("Q" if board[i][j] == 1 else ".", end=" ")

print()

print("\n")

def is\_safe(board, row, col, N):

"""Check if placing a queen at (row, col) is safe."""

# Check this row on the left

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on the left

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on the left

for i, j in zip(range(row, N), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_nqueens\_util(board, col, N):

"""Use backtracking to place queens."""

if col >= N: # All queens placed

print\_solution(board, N)

return True

res = False

for i in range(N):

if is\_safe(board, i, col, N):

board[i][col] = 1 # Place queen

res = solve\_nqueens\_util(board, col + 1, N) or res

board[i][col] = 0 # Backtrack

return res

def solve\_nqueens(N):

"""Main function to solve N Queens CSP problem."""

board = [[0] \* N for \_ in range(N)]

if not solve\_nqueens\_util(board, 0, N):

print("No solution exists")

else:

print("All possible solutions printed above.")

# Driver code

N = int(input("Enter the number of queens: "))

solve\_nqueens(N)

**Output:**

Enter the number of queens: 4

. . Q .

Q . . .

. . . Q

. Q . .

. Q . .

. . . Q

Q . . .

. . Q .

All possible solutions printed above.

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 30: N Queens using Hill Climbing**

**Aim**:  
To solve the N-Queens problem using the Hill Climbing algorithm.

**Algorithm**:

1. Start with a random configuration of queens.
2. Evaluate heuristic (number of attacking pairs).
3. Move to neighbor with lower heuristic.
4. Repeat until no improvement possible.
5. If heuristic = 0 → solution found. Else → local minimum

**Program in Python**:

import random

def print\_board(board):

"""Helper function to print the board."""

N = len(board)

for i in range(N):

row = ""

for j in range(N):

row += "Q " if board[i] == j else ". "

print(row)

print("\n")

def calculate\_conflicts(board):

"""Count the number of pairs of queens attacking each other."""

N = len(board)

conflicts = 0

for i in range(N):

for j in range(i + 1, N):

# Same column or same diagonal

if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):

conflicts += 1

return conflicts

def get\_best\_neighbor(board):

"""Find the best neighbor by moving one queen in its column."""

N = len(board)

best\_board = board[:]

best\_conflicts = calculate\_conflicts(board)

for col in range(N):

for row in range(N):

if board[col] != row:

new\_board = board[:]

new\_board[col] = row

conflicts = calculate\_conflicts(new\_board)

if conflicts < best\_conflicts:

best\_conflicts = conflicts

best\_board = new\_board[:]

return best\_board, best\_conflicts

def hill\_climbing(N, max\_restarts=1000):

"""Hill Climbing with random restarts to solve N-Queens."""

for restart in range(max\_restarts):

# Generate random initial board

board = [random.randint(0, N-1) for \_ in range(N)]

conflicts = calculate\_conflicts(board)

while True:

neighbor, neighbor\_conflicts = get\_best\_neighbor(board)

if neighbor\_conflicts >= conflicts:

break # Local minimum reached

board, conflicts = neighbor, neighbor\_conflicts

if conflicts == 0:

print(f"Solution found after {restart} restarts:\n")

print\_board(board)

return board

print("No solution found with given restarts.")

return None

# Driver Code

N = int(input("Enter number of queens: "))

hill\_climbing(N)

**Output:**

Enter number of queens: 4

Solution found after 2 restarts:

. Q . .

. . . Q

Q . . .

. . Q .

[1, 3, 0, 2]

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 31: Propositional Logic Evaluation using SymPy**

**Aim**:  
To evaluate propositional logic expressions using SymPy.

**Algorithm**:

1. Import symbols and And/Or/Not/Implies/Equivalent from sympy.logic.boolalg.
2. Define logical variables.
3. Construct a propositional logic expression.
4. Simplify or evaluate the expression.
5. Print result.

**Installing pytholog**:

!pip install pytholog

Requirement already satisfied: pytholog in c:\users\shail\appdata\local\programs\python\python39\lib\site-packages (2.4.1)

Requirement already satisfied: more-itertools in c:\users\shail\appdata\local\programs\python\python39\lib\site-packages (from pytholog) (9.0.0)



**Logical Sentence:**

import pytholog as pl

from sympy import \*

from sympy.logic.inference import satisfiable

rain = Symbol("rain")

hagrid = Symbol("hagrid")

dumbledore = Symbol("dumpledore")

logical\_sentence = And(rain, hagrid)

print(logical\_sentence)

print(logical\_sentence.subs({rain: True, hagrid: False}))

print(satisfiable(logical\_sentence))

**Output:**

hagrid & rain

False

{hagrid: True, rain: True}



**Implication 1:**

implication\_logic = Implies(Not(rain), hagrid)

print(implication\_logic)

print(satisfiable(implication\_logic))

**Output:**

Implies(~rain, hagrid)

{rain: True, hagrid: True}

****

**Knowledge base:**

knowledge\_base = And(

Implies(Not(rain), hagrid),

Or(hagrid, dumbledore),

Not(And(hagrid, dumbledore)),

hagrid

)

print(knowledge\_base)

models = satisfiable(knowledge\_base, all\_models=True)

for model in models:

print(model)

**Output:**

hagrid & (dumpledore | hagrid) & (Implies(~rain, hagrid)) & ~(dumpledore & hagrid)

{hagrid: True, rain: True, dumpledore: False}

{hagrid: True, rain: False, dumpledore: False}

**Implication 2: (¬p ∧ q) → r**

**"If it is not rainy and you are happy, then you will play football."**

from sympy.logic.inference import valid

p = Symbol("rainy")

q = Symbol("happy")

r = Symbol("play football")

p1 = Implies(And(Not(p),q),r)

print(valid(p))

**Output:**

False

**Implication 3: [(¬p ∧ q) ∧ (r → p) ∧ (¬r → s) ∧ (s → t)] → t**

**“If it's not rainy in the afternoon and colder than yesterday, and we go swimming only if it's sunny, and if we don't go swimming we take a canoe trip, and if we take a canoe trip we're home by sunset — then we will be home by sunset.”**

from sympy.logic.inference import valid

p = Symbol("rain\_afternoon")

q = Symbol("colder\_than\_yesterday")

r = Symbol("go\_swimming")

s = Symbol("take\_a\_canoe\_trip")

t = Symbol("at\_home\_by\_sunset")

p1 = And(Not(p),q) #It’s not sunny and it’s colder than yesterday”]

p2 = Implies(r,p) # “We will go swimming only if it’s sunny.”

p3 = Implies(Not(r),s) #“If we don’t go swimming then we will take a canoe trip.”

p4 = Implies(s,t) #“If we take a canoe trip, then we will be home by sunset

result = valid(Implies(And(p1,p2,p3,p4),t))

print(result)

**Output:**

True

**Result**:  
The program is successfully executed.

**Date:**

**Experiment 32: Propositional Logic – Validate Logical Rule**

**Aim**:  
To verify if a logical rule is valid using SymPy’s simplify\_logic or truth tables

**Algorithm**:

1. Define logical symbols.
2. Construct the implication to be validated.
3. Simplify expression.
4. If always True → logically valid.

**Installing pytholog**:

!pip install pytholog

Requirement already satisfied: pytholog in c:\users\shail\appdata\local\programs\python\python39\lib\site-packages (2.4.1)

Requirement already satisfied: more-itertools in c:\users\shail\appdata\local\programs\python\python39\lib\site-packages (from pytholog) (9.0.0)

**Knowledge Base**:

import pytholog as pl

new\_kb = pl.KnowledgeBase("flavor")

new\_kb(["likes(noor, sausage)",

"likes(melissa, pasta)",

"likes(dmitry, cookie)",

"likes(nikita, sausage)",

"likes(assel, limonade)",

"food\_type(gouda, cheese)",

"food\_type(ritz, cracker)",

"food\_type(steak, meat)",

"food\_type(sausage, meat)",

"food\_type(limonade, juice)",

"food\_type(mojito, juice)",

"food\_type(cookie, dessert)",

"flavor(sweet, dessert)",

"flavor(savory, meat)",

"flavor(savory, cheese)",

"flavor(sweet, juice)",

"food\_flavor(X, Y) :- food\_type(X, Z), flavor(Y, Z)",

"dish\_to\_like(X, Y) :- likes(X, L), food\_type(L, T), flavor(F, T), food\_flavor(Y, F), neq(L, Y)"])

**Query 1:**

new\_kb.query(pl.Expr("likes(noor, sausage)"))

**Output 1:**

['Yes']

**Query 2:**

new\_kb.query(pl.Expr("likes(noor, pasta)"))

**Output 2:**

['No']

**Query 3:**

from time import time

start = time()

print(new\_kb.query(pl.Expr("food\_flavor(What, sweet)")))

print(time() - start)

**Output 3:**

[{'What': 'cookie'}, {'What': 'limonade'}, {'What': 'mojito'}]

0.0018224716186523438

**Query 4:**

start = time()

print(new\_kb.query(pl.Expr("dish\_to\_like(noor, What)")))

print(time() - start)

**Output 4:**

[{'What': 'gouda'}, {'What': 'steak'}]

0.0009131431579589844

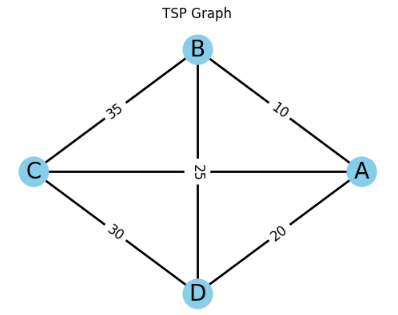
**Result**:  
The program is successfully executed.

**Date:**

**Experiment 33: Brute Force – Travelling Salesman Problem (TSP)**

**Aim**:  
To solve TSP using brute force permutation method.

**Algorithm**:

1. Represent cities as graph with distance matrix.
2. Generate all possible permutations of cities.
3. Compute tour cost for each permutation.
4. Return path with minimum cost.

**Program in Python**:

from itertools import permutations

# Distance matrix

dist = {

('A','B'): 10, ('A','C'): 15, ('A','D'): 20,

('B','A'): 10, ('B','C'): 35, ('B','D'): 25,

('C','A'): 15, ('C','B'): 35, ('C','D'): 30,

('D','A'): 20, ('D','B'): 25, ('D','C'): 30

}

cities = ['A','B','C','D']

start = 'A'

min\_cost = float('inf')

best\_path = []

for perm in permutations([c for c in cities if c != start]):

path = [start] + list(perm) + [start]

cost = sum(dist[(path[i], path[i+1])] for i in range(len(path)-1))

if cost < min\_cost:

min\_cost = cost

best\_path = path

print("Best Path:", best\_path)

print("Minimum Cost:", min\_cost)

**Output:**

Best Path: ['A', 'B', 'D', 'C', 'A']

Minimum Cost: 80

**Result**:  
The program is successfully executed.

# Viva Questions

### Python Basics

1. **Q:** What is Python?  
    **A:** Python is a high-level, interpreted, object-oriented programming language with dynamic semantics.
2. **Q:** How is Python different from C?  
    **A:** Python is interpreted, dynamically typed, and has built-in data structures; C is compiled, statically typed, and low-level.
3. **Q:** What are Python’s key features?  
    **A:** Easy syntax, dynamic typing, portability, libraries, and support for OOP and functional programming.
4. **Q:** What is the difference between print() and return?  
    **A:** print() displays output to the console, while return sends values back from a function.
5. **Q:** What are Python data types?  
    **A:** Numeric (int, float, complex), Boolean, Sequence (list, tuple, string), Set, Dictionary.

### Operators and Control Structures

1. **Q:** What are Python arithmetic operators?  
    **A:** +, -, \*, /, %, //, \*\*.
2. **Q:** What is the difference between == and is?  
    **A:** == checks values; is checks object identity (memory address).
3. **Q:** What are Boolean operators?  
    **A:** and, or, not.
4. **Q:** What is the use of if-elif-else?  
    **A:** To implement decision-making with multiple conditions.
5. **Q:** What is the difference between while and for loops?  
    **A:** for iterates over a sequence, while executes until a condition is false.

### Data Structures

1. **Q:** What is the difference between list and tuple?  
    **A:** Lists are mutable, tuples are immutable.
2. **Q:** What is a set in Python?  
    **A:** An unordered collection of unique elements.
3. **Q:** How is dictionary different from list?  
    **A:** Dictionary stores key-value pairs, list stores ordered elements.
4. **Q:** What is list comprehension?  
    **A:** A concise way to create lists using [expression for item in iterable if condition].
5. **Q:** How do you access elements of a list?  
    **A:** Using indexing (list[0]) and slicing (list[1:3]).

### Functions & OOP

1. **Q:** What is a function?  
    **A:** A reusable block of code defined using def.
2. **Q:** What is recursion?  
    **A:** When a function calls itself.
3. **Q:** What are the advantages of functions?  
    **A:** Reusability, modularity, readability.
4. **Q:** What is encapsulation?  
    **A:** Wrapping variables and methods into a class to protect data.
5. **Q:** What are class and object?  
    **A:** Class is a blueprint; object is an instance of the class.

### Search Algorithms

1. **Q:** What is DFS?  
    **A:** Depth First Search explores nodes along a path before backtracking.
2. **Q:** What is BFS?  
    **A:** Breadth First Search explores nodes level by level.
3. **Q:** Difference between DFS and BFS?  
    **A:** DFS uses stack (or recursion), BFS uses queue.
4. **Q:** What is Uniform Cost Search?  
    **A:** A search strategy that expands the least-cost node first.
5. **Q:** What is Depth Limited Search?  
    **A:** A variant of DFS with a predefined depth limit.

### Heuristic Search

1. **Q:** What is Greedy Best First Search?  
    **A:** A heuristic search that expands nodes with the lowest estimated cost to goal (h(n)).
2. **Q:** What is A\* algorithm?  
    **A:** A search algorithm using both path cost (g(n)) and heuristic (h(n)): f(n) = g(n) + h(n).
3. **Q:** Why is A\* better than GBFS?  
    **A:** A\* guarantees the optimal path if the heuristic is admissible.
4. **Q:** What is a heuristic?  
    **A:** An estimate of the cost to reach the goal from a node.
5. **Q:** What is the drawback of GBFS?  
    **A:** It can get stuck in local minima and may not find the optimal solution.

### CSP & Optimization

1. **Q:** What is CSP?  
    **A:** Constraint Satisfaction Problem involves finding values for variables satisfying constraints.
2. **Q:** What is the N-Queens problem?  
    **A:** Placing N queens on a chessboard such that no two attack each other.
3. **Q:** Which algorithm can solve N-Queens?  
    **A:** Backtracking, Hill Climbing, CSP solvers.
4. **Q:** What is Hill Climbing?  
    **A:** A local search algorithm that continuously moves towards better solutions.
5. **Q:** What is the drawback of Hill Climbing?  
    **A:** It can get stuck in local maxima or plateaus.

### Logic and AI

1. **Q:** What is propositional logic?  
    **A:** Logic using propositions that are either True or False.
2. **Q:** What is Modus Ponens?  
    **A:** If P → Q and P are true, then Q is true.
3. **Q:** What is SymPy in Python?  
    **A:** A library for symbolic mathematics including logic evaluation.
4. **Q:** How do you validate a logical rule in SymPy?  
    **A:** By simplifying the expression and checking if result is True.
5. **Q:** What is the difference between propositional and predicate logic?  
    **A:** Propositional deals with statements; predicate includes quantifiers (∀, ∃).

### TSP & Advanced

1. **Q:** What is the Travelling Salesman Problem (TSP)?  
    **A:** Finding the shortest path visiting all cities exactly once and returning to start.
2. **Q:** What is brute force TSP?  
    **A:** Checking all permutations of cities and choosing the shortest.
3. **Q:** What is the drawback of brute force TSP?  
    **A:** Time complexity grows factorially with number of cities.
4. **Q:** What is the time complexity of brute force TSP?  
    **A:** O(n!).
5. **Q:** Can heuristic methods solve TSP faster?  
    **A:** Yes, but they may not always find the exact optimal path.

### General Viva

1. **Q:** What is difference between algorithm and program?  
    **A:** Algorithm is a step-by-step procedure; program is its implementation in a language.
2. **Q:** What is a data structure?  
    **A:** A way of organizing and storing data efficiently.
3. **Q:** What is the use of AI search algorithms?  
    **A:** To find optimal or feasible solutions in state space problems.
4. **Q:** What are examples of real-life AI search?  
    **A:** Pathfinding in maps, puzzle solving, scheduling.
5. **Q:** Why do we study both Python basics and AI algorithms in this lab?  
    **A:** Python provides the foundation; AI algorithms apply these concepts to problem-solving.