1.A*

```
import heapq

def a_star(start, goal, graph):
    open_set = [(0, start)]
    closed_set = set()
    g = {start: 0}
    parents = {start: start}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = [current]
            while current != start:
                current = parents[current]
                path.append(current)
            print('Path found:', path[::-1])
            return

        closed_set.add(current)

        for neighbor, weight in graph.get(current, []):
            if neighbor not in closed_set:
                tentative_g = g[current] + weight
                if neighbor not in g or tentative_g < g[neighbor]:
                    g[neighbor] = tentative_g
                    parents[neighbor] = current
                    heapq.heappush(open_set, (tentative_g + heuristic(neighbor, goal), neighbor))

    print('Path does not exist!')

def heuristic(n, goal):
    h_dist = {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0}
    return h_dist.get(n, float('inf'))

graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9), ('A', 2)],
    'C': [('B', 1)],
    'E': [('D', 6), ('A', 3)],
    'D': [('G', 1), ('E', 6)],
    'G': [('B', 9), ('D', 1)]
}

start_node = input("Enter Start Node (A, B, C, D, E, G): ").upper()
stop_node = input("Enter Stop Node (A, B, C, D, E, G): ").upper()

if start_node in graph_nodes and stop_node in graph_nodes:
    a_star(start_node, stop_node, graph_nodes)
else:
    print("Invalid start or stop node.")
```

2.BFS

```
import queue
def bfs_with_queue(graph, start):
    visited = []
    q = queue.Queue()
    q.put(start)

    while not q.empty():
        node = q.get()
        if node not in visited:
            visited.append(node)
            print(node, end=' ')

        for neighbors in graph[node]:
            if neighbors not in visited:
                q.put(neighbors)

graph = {}
node=[x for x in input("Enter nodes :").split()]
for i in node:
    neighbor = input(f"Enter the neighbors of node {i}: ").split()
    graph[str(i)] = neighbor
start_node = input("Enter the starting node for BFS: ")
if start_node in graph:
    print("Following is the Breadth-First Search:")
    bfs_with_queue( graph, start_node)
else:
    print("Starting node not found in the graph.")
```

3.DFS

```
import queue
def bfs_with_queue(graph, start):
    visited = []
    q = queue.Queue()
    q.put(start)

    while not q.empty():
        node = q.get()
        if node not in visited:
            visited.append(node)
            print(node, end=' ')

        for neighbors in graph[node]:
            if neighbors not in visited:
                q.put(neighbors)

graph = {}
node=[x for x in input("Enter nodes :").split()]
for i in node:
    neighbor = input(f"Enter the neighbors of node {i}: ").split()
    graph[str(i)] = neighbor
```

```
start_node = input("Enter the starting node for BFS: ")
print("Following is the Breadth-First Search:")
bfs_with_queue( graph, start_node)
```

4.Greedy BFS

```
from queue import PriorityQueue

def greedy_best_first_search(graph, start, goal, heuristic):
    priority_queue = PriorityQueue()
    priority_queue.put((heuristic(start, goal), start, [start]))  # Include the path as a third element in
the tuple
    visited = set()

    while not priority_queue.empty():
        current_heuristic, current_node, current_path = priority_queue.get()

        if current_node == goal:
            print("Goal reached:", current_node)
            print("Path:", current_path)
            return True

        visited.add(current_node)
      #  print("The visited list is:",visited)

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                new_path = current_path + [neighbor]  # Extend the path
                priority_queue.put((heuristic(neighbor, goal), neighbor, new_path))

    print("Goal not reached")
    return False

# Example usage
graph = {
    'A': {'B': 2, 'C': 7},
    'B': {'D':2},
    'C': {'E':4,'D':2},
    'D': {'E':5},
    'E': {}

}

def heuristic(node, goal):
    heuristic_values = {'A': 11, 'B': 8, 'C': 4, 'D': 5, 'E': 0}
    return heuristic_values[node]

start_node = 'A'
goal_node = 'E'

greedy_best_first_search(graph, start_node, goal_node, heuristic)
```

5.Non-linear regression

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Define the quadratic functzion
def quadratic_function(x, a, b, c):
    return a * x**2 + b * x + c
# Load data from CSV file
data = pd.read_csv('non_reg_data.csv')
print(data.head())
x = data['Year'].values  # Feature (independent variable)
y = data['Value'].values  # Target variable (dependent variable)
# Fit the quadratic function to the data
initial_guess = [1, 1, 1]  # Initial guess for the parameters (a, b, c)
params, covariance = curve_fit(quadratic_function, x, y, initial_guess)
# Extract the fitted parameters (regression coefficients)
a, b, c = params
# Generate predicted values using the fitted parameters
y_fit = quadratic_function(x, a, b, c)
print(f'Regression Equation: y = {a:.2f}x^2 + ({b:.2f})x + {c:.2f}')
# Predict future values
x_new = 2023  # Replace this with the desired future year
y_predicted = quadratic_function(x_new, a, b, c)
print(f'Predicted y value for x={x_new}: {y_predicted}\n')
# Plot the original data and the fitted curve
plt.scatter(x, y, label='Original Data')
plt.plot(x, y_fit, color='red', label='Quadratic Fit')
plt.legend()
plt.xlabel('Year')
plt.ylabel('Value')
plt.title('Quadratic Nonlinear Regression')
plt.show()
```

6.Simple linear regression

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

data = pd.read_csv('Salary_dataset.csv')
data.head()

*) x = data[['YearsExperience']].values # Feature
y = data['Salary'].values # Target variable
```

```
*)X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 1/3, random_state = 1)
regres = LinearRegression()
regres.fit(X_train, y_train)
y_pred = np.round(regres.predict(X_test), 2)
y_pred

*) y_test
*) score = r2_score(y_test, y_pred)
print("The accuracy of our model is {}%".format(round(score, 2) *100))

*) year=int(input("Enter the year of experience to predict salary : "))
print(f"The predicted salary for given {year} years : ",np.round(regres.predict([[year]]),2))

print(f"Regression Coefficient: {regres.coef_[0]:.2f}")
print(f"Intercept : {regres.intercept_:.2f}")

*) # Visualizing both Training and Test set results on a single plot
plt.scatter(X_train, y_train, color='red', label='Training set')
plt.scatter(X_test, y_test, color='blue', label='Test set')
plt.plot(X_train, regres.predict(X_train), color='green', linewidth=2, label='Regression line')
plt.title('Salary vs Experience')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.legend()
plt.show()
```

7.N-queens

```
# Taking the number of queens as input from the user
print("Enter the number of queens")
N = int(input())

# Here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0] * N for _ in range(N)]


def attack(i, j):
    # Checking vertically and horizontally
    for k in range(0, N):
        if board[i][k] == 1 or board[k][j] == 1:
            return True
    # Checking diagonally
    for k in range(0, N):
        for l in range(0, N):
            if (k + l == i + j) or (k - l == i - j):
                if board[k][l] == 1:
                    return True
    return False


def N_queens(n):
    if n == 0:
        return True
    for i in range(0, N):
        for j in range(0, N):
            if (not attack(i, j)) and (board[i][j] != 1):
                board[i][j] = 1
                if N_queens(n - 1):
                    return True
                board[i][j] = 0
    return False


N_queens(N)

# Print the board with 'Q' for queens and '.' for empty squares
for row in board:
    print(" ".join("Q" if square == 1 else "." for square in row))
```