LAB 1:

Breadth-First Search (BFS) Algorithm Steps:

1. Start by creating a queue and insert the starting state.
2. Create a set to record states that have been explored.
3. Continue if the queue has elements, otherwise, follow steps 4-8.
4. Remove and examine the state at the front of the queue.
5. If this state is the target, deliver the solution.
6. Create all potential subsequent states from this state.
7. If a subsequent state hasn't been seen before, note it as seen and add it to the queue.
8. Should the queue be depleted without finding a solution, indicate failure.

Depth-First Search (DFS) Algorithm Steps:

1. Begin with a stack, placing the initial state inside.
2. Establish a set to log states that have been visited.
3. Proceed if the stack isn't empty, otherwise, execute steps 4-8.
4. Take and inspect the state at the top of the stack.
5. If this state is the objective, provide the solution.
6. Develop all potential subsequent states from this state.
7. If a subsequent state is unvisited, mark it as visited and place it on the stack.
8. If you exhaust the stack without discovering a solution, declare failure.

Uniform Cost Search (UCS) Algorithm Steps:

1. Initiate a priority queue, adding the initial state with zero cost.
2. Formulate a set to track states that have been visited.
3. As long as there are elements in the priority queue, continue; if not, follow steps 4-8.
4. Take out the lowest-cost state from the priority queue for examination.
5. If this state is your goal, return with the solution.
6. Generate all possible subsequent states from this state with their respective costs.
7. For any subsequent state that is either unvisited or found via a less costly route, update its cost and include it in the priority queue.
8. Declare failure if no solution emerges after emptying the priority queue.

Depth-Limited Search (DLS) Algorithm Steps:

1. Set up a stack with the initial state at depth zero.
2. Create a set to keep tabs on visited states.
3. Continue if there's content in the stack; if not, proceed with steps 4-9.
4. Remove and evaluate the topmost state from the stack.
5. Return with the solution if this state matches your goal.
6. Move on to another iteration if you've reached your depth limit at this point.
7. Formulate all potential subsequent states from this current one.
8. For each new state that hasn't been explored, mark it as visited and add it to your stack with an increased depth count.
9. Announce failure if you deplete your stack without finding a solution.

LAB 2:

A* Algorithm Procedure:

1. Initialize sets and dictionaries:

- Open set: Add the starting node.
- Closed set: Empty.
- g: Distance from the start node.
- parents: Parent of each node.
- Set g[start] to 0 and parents[start] to itself.

2. Main Loop (until the open set is empty):

- Select node n with the lowest f() = g[n] + heuristic(n) from the open set.
- If n is the goal or has no neighbors, continue.

3. Process neighbors of n:

- For each neighbor (m, weight):
  - If m is not in the open or closed set:
    - Add m to the open set, set parents[m] to n, and update g[m].
  - Else if g[m] > g[n] + weight:
    - Update g[m] and change parents[m] to n.
    - If m is in the closed set, move it to the open set.

4. Handle no selected node:

- If n is None, return "Path does not exist".

5. Check goal:

- If n is the goal, reconstruct and return the path using parents.

6. Move n to closed set:

- Remove n from the open set and add it to the closed set.

7. Final check:

- If the open set is empty and no path was found, return "Path does not exist".

Best First Search Algorithm:

1. Initialize the open set:
- Add the starting node to the open set.

2. Initialize the closed set:
- The closed set starts as empty.

2. Initialize dictionaries:
- `parents` dictionary: Stores the parent of each node. - Set the start node's parent to itself.

4. Main Loop:
- Continue until the open set is empty:

- Set `n` to the node with the lowest heuristic value in the open set.

5. Check if the current node is the stop node: - If `n` is the stop node:

- Reconstruct the path from the stop node to the start node using the `parents` dictionary. - Print and return the path.

6. Process each neighbor of `n`:
- For each neighbor `(m, weight)` of `n`:

- If `m` is not in the open set or closed set: - Add `m` to the open set.
- Set `m`'s parent to `n`.

7. Move `n` to the closed set:
- Remove `n` from the open set. - Add `n` to the closed set.

8. Handle case where no path is found:
- If the open set is empty and no path was found, print "Path does not exist" and return `None`.

LAB 3:

- **Algorithms BFS Water Jug Problem Algorithm Steps**

- **Step 1:**
  Import deque from collections.
  Define bfs_jugs function with capacity_a, capacity_b, and target as parameters.
  Begin with an empty deque named queue.
  Create a set named visited to track explored states.
  Insert the starting state into the queue and mark it visited, indicating both jugs are empty with zero steps and no actions taken yet.

- **Step 2:** BFS Iteration
  Continue if the queue has elements. For each iteration:
  Remove the first state from the queue, denoted as ((current_a, current_b), steps, sequence), where current_a and current_b are the water levels in jugs A and B, steps is the count of actions taken, and sequence lists the operations done.

- **Step 3:** Check for Solution
  If either current_a or current_b equals the target, return the sequence as the solution.
  Step 4: State Expansion Prepare an empty list for potential new states.

- **Step 5:** Execute Operations Perform each operation below, creating a new state. If it's unvisited, add to next_states with increased step count and updated sequence:

  Fill Jug A to capacity_a.
  Fill Jug B to capacity_b.

Empty Jug A completely.
Empty Jug B completely.
Transfer water from A to B until A is empty or B is full. Transfer water from B to A until B is empty or A is full.

Step 6: Queue Next States

Add each new state from next_states to the queue for subsequent exploration.

Step 7: No Solution Found

If the queue empties without finding a solution, return None as an indication that the target measurement isn't achievable.

Algorithm for DFS Water Jug Problem

Step 1: Start with an empty stack.
Create a set named visited to record explored states.

Place the initial state ((0, 0), 0, []) on the stack and mark it visited, signifying both jugs are empty with no steps or actions taken.

Step 2: DFS Iteration
As long as the stack isn't empty, carry out these steps:

Take the top state from the stack, represented as ((current_a, current_b), steps, sequence), where current_a and current_b are the volumes in jugs A and B, steps is the action count, and sequence is the operations list.

Step 3: Solution Verification

If current_a or current_b matches the target, return the sequence as the solution.

Step 4: State Development

Prepare an empty list for new potential states.

Step 5: Perform Operations

For each operation below, if it leads to an unvisited state, add it to next_states with an incremented step count and an updated sequence:

Fill Jug A to capacity_a.
Fill Jug B to capacity_b.
Drain Jug A entirely.
Drain Jug B entirely.
Transfer from A to B until A is empty or B is full. Transfer from B to A until B is empty or A is full.
Step 6: Stack Next States :Push each state from next_states onto the stack for more exploration.

Step 7: Inconclusive Outcome

If no solution emerges and the stack depletes, return None to show that it’s impossible to measure the target amount of water.

LAB 4:

1.Reading the CSV file:
Read the CSV file named “Iris.csv” located at “D:\Iris.csv” into a Pandas DataFrame. Store the DataFrame in the variable sample_dataframe_df.

2.Displaying the DataFrame type:
Print the type of sample_dataframe_df.

3.Setting display options:
Set the maximum number of displayed columns to 5 using pd.set_option('display.max_columns', 5).

4.Displaying the first 5 rows:
Print the first 5 rows of the DataFrame using sample_dataframe_df.head(5).

5.Displaying column names:
Print the column names using sample_dataframe_df.columns.

6.Transposing the first 5 rows:
Transpose the first 5 rows of the DataFrame using sample_dataframe_df.head(5).transpose().

7.Displaying the shape of the DataFrame:
Print the shape (number of rows and columns) of the DataFrame using sample_dataframe_df.shape.

8.Slicing the first 10 rows:
Print the first 10 rows of the DataFrame using sample_dataframe_df[0:10].

9.Slicing the last 5 rows:
Print the last 5 rows of the DataFrame using sample_dataframe_df[-5:].

10.Accessing the ‘SepalLengthCm’ column:
Print the values in the ‘SepalLengthCm’ column using sample_dataframe_df['SepalLengthCm'].

11.Counting species occurrences:
Print the count of each unique value in the ‘Species’ column using sample_dataframe_df.Species.value_counts().

12.Filtering rows based on ‘SepalLengthCm’:
Print rows where the ‘SepalLengthCm’ value is greater than 1 using sample_dataframe_df[sample_dataframe_df['SepalLengthCm'] > 1].

13.Dropping the ‘Id’ column:
Drop the ‘Id’ column from the DataFrame in-place using sample_dataframe_df.drop('Id', inplace=True, axis=1).

14.Listing remaining column names:
Print the updated list of column names using list(sample_dataframe_df.columns).

LAB 5:

Import numpy as np

- Import pandas as pd
- Import stats from scipy
- Import pyplot from matplotlib as plt
- Import seaborn as sns
- Enable inline plotting for Jupyter notebooks using %matplotlib inline
- Read the CSV file from the specified path into a DataFrame df.

Descriptive Statistics:

```
mean_value = df.mean()
median_value = df.median()
mode_value = df.mode().iloc[0]
std_deviation = df.std()
variance = df.var()
percentile_95 = df.quantile(0.95)
```

Generate Random Values:

```
bd = np.random.uniform(0, 10, 350)
```

Visualizations:

```
sns.barplot(x='Kms Driven', y='Selling Price', data=df)
plt.show()

hue using Seaborn
sns.barplot(x='Kms Driven', y='Selling Price', hue='Year',
data=df)
plt.show()

# Histogram
plt.hist(df['Selling Price'])
plt.show()

# Histogram of 'Selling Price' with 5 bins using Matplotlib
plt.hist(df['Selling Price'], bins=5)
plt.show()

# Box plot of 'Selling Price' using Seaborn
```

```
sns.boxplot(y='Selling Price', data=df)
plt.show()

# Scatter plot of 'Selling Price' vs. 'Kms Driven' using
Matplotlib
plt.scatter(df['Kms Driven'], df['Selling Price'])
plt.xlabel('Kms Driven')
plt.ylabel('Selling Price')
plt.show()

# Pair plot of the entire DataFrame using Seaborn
sns.pairplot(df)
plt.show()

# Heatmap with annotations using Seaborn
features_of_interest = df[['Selling Price', 'Kms Driven',
'Year']]
correlation_matrix = features_of_interest.corr()
sns.heatmap(correlation_matrix, annot=True)
plt.show()
```

LAB 6:

Algorithm for K-Nearest Neighbors Classification

1. **Import necessary libraries**:

o Import KNeighborsClassifier from sklearn.neighbors.
o Import train_test_split from sklearn.model_selection. o Import pandas as pd.

2. **Load the Iris dataset**:

o Read the CSV file into a DataFrame df.
o Print the first 5 rows of the DataFrame.
o Display the value counts for the 'Species' column.

3. **Prepare the feature and target variables**:

o Drop the 'Id' and 'Species' columns from df to create the feature set x. o Set the 'Species' column as the target variable y.
o Print the first 5 rows of x and y.

4. **Split the dataset**:

o Use train_test_split to split x and y into training and testing sets with an 80-20 split.

o Set random_state to 80 for reproducibility.

5. **Train the K-Nearest Neighbors model**:

o Instantiate the KNeighborsClassifier with n_neighbors set to 5. o Fit the model using the training data (train_x and train_y).

6. **Evaluate the model**:
o Compute and print the accuracy score of the model on the test set (test_x and

test_y).

Algorithm for Decision Tree Classification

1. **Import necessary libraries**:

o Import numpy as np.
o Import train_test_split from sklearn.model_selection. o Import pandas as pd.
o Import DecisionTreeClassifier from sklearn.tree.
o Import accuracy_score from sklearn.metrics.

2. **Load the Loan Repayment dataset**:
o Read the CSV file into a DataFrame df.

3. **Prepare the feature and target variables**:

o Drop the 'Result' column from df to create the feature set x.

o Set the 'Result' column as the target variable y.

4. **Split the dataset**:

o Use train_test_split to split x and y into training and testing sets with a 20-80 split.

o Set random_state to 100 for reproducibility.

5. **Train the Decision Tree model**:

o Instantiate the DecisionTreeClassifier.

o Fit the model using the training data (train_x and train_y).

6. **Make predictions and evaluate the model**:

o Use the trained model to predict the target variable on the test set (test_x).
o Print the predicted values.
o Compute and print the accuracy score of the model using the actual and predicted

values (test_y and predict_y).

-