Campus AI Practicals

A Comprehensive Guide to Artificial Intelligence Algorithms and Implementations

Table of Contents

Introduction

This document presents three fundamental practicals in Artificial Intelligence and Computer Science, designed for academic learning and practical implementation. Each practical includes comprehensive theoretical background, algorithmic explanations, and hands-on implementation using Python.

The practicals cover:

- Essential Python libraries for AI development
- Graph traversal algorithms: BFS and DFS
- Practical applications in problem-solving and data structures

Practical 1: Python Libraries for AI Development

Objective

To understand and implement basic usage of essential Python libraries commonly used in Artificial Intelligence, Machine Learning, and Web Development.

Theory

Python has become the de facto language for AI development due to its rich ecosystem of libraries. Understanding these libraries is crucial for any AI practitioner:

Core Libraries Overview:

1. LangChain & LangGraph

- **LangChain**: Framework for developing applications powered by language models
- **LangGraph**: Extension for building stateful, multi-actor applications with LLMs
- **Use Cases**: Chatbots, document analysis, conversational AI

2. Data Science Libraries

- **Pandas**: Data manipulation and analysis library
- **NumPy**: Fundamental package for scientific computing
- **Matplotlib**: Comprehensive library for creating static, animated, and interactive visualizations

3. Machine Learning Libraries

- **scikit-learn**: Simple and efficient tools for predictive data analysis
- **TensorFlow**: End-to-end open source platform for machine learning

4. Computer Vision

- **OpenCV**: Library of programming functions mainly aimed at real-time computer vision

5. Web Development Frameworks

- **Streamlit**: Framework for creating data apps
- **FastAPI**: Modern, fast web framework for building APIs
- **Django**: High-level Python web framework

Implementation

```
# LangChain Example (requires installation and API keys)
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-003")
response = llm("What is LangChain?")
print(response)

# Matplotlib Example
import matplotlib.pyplot as plt
plt.plot([1,2,3],[4,5,6])
plt.title("Matplotlib Example")
plt.show()

# Pandas Example
import pandas as pd
df = pd.DataFrame({'A':[1,2],'B':[3,4]})
print(df)

# NumPy Example
import numpy as np
arr = np.array([1,2,3])
```

```
print(arr * 2)

# scikit-learn Example
from sklearn.linear_model import LinearRegression
model = LinearRegression()
X = [[0],[1],[2]]
y = [0,1,2]
model.fit(X, y)
print(model.predict([[3]]))

# TensorFlow Example
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
print(tf.add(a, b))
```

Learning Outcomes

- Understanding of essential Python libraries for AI development
- Basic implementation skills for various frameworks
- Foundation for advanced AI and ML projects

Practical 2: Breadth-First Search (BFS) Algorithm

Objective

To understand, implement, and analyze the Breadth-First Search algorithm for graph traversal and pathfinding problems.

Theory

What is Breadth-First Search?

Breadth-First Search (BFS) is a graph traversal algorithm that explores vertices in layers, visiting all vertices at the current depth before moving to vertices at the next depth level. It uses a queue data structure to maintain the order of exploration.

Algorithm Characteristics:

- **Time Complexity**: O(V + E) where V is vertices and E is edges
- **Space Complexity**: O(V) for the queue and visited array
- **Completeness**: Always finds a solution if one exists
- **Optimality**: Finds the shortest path in unweighted graphs

BFS Algorithm Steps:

1. Start with a source vertex and mark it as visited
2. Add the source vertex to a queue
3. While the queue is not empty:
    - Dequeue a vertex from the front of the queue
    - For each unvisited neighbor of the current vertex:
        - Mark the neighbor as visited
        - Add the neighbor to the queue
        - Set the parent of the neighbor (for path reconstruction)

Applications:

- Finding shortest path in unweighted graphs
- Level-order traversal of trees
- Web crawling
- Social network analysis
- GPS navigation systems

Implementation

Graph Visualization Setup

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a sample graph
G = nx.Graph()
G.add_nodes_from(range(7))

edges = [
    (0, 1), (1, 2), (2, 3), (3, 4),
    (4, 5), (5, 6), (0, 4), (2, 5)
]

G.add_edges_from(edges)
pos = nx.spring_layout(G)

# Visualize the graph
plt.figure(figsize=(8, 8))
nx.draw_networkx_nodes(G, pos, node_color='lightgreen', node_size=700)
nx.draw_networkx_edges(G, pos, width=2, edge_color='gray')
nx.draw_networkx_labels(G, pos, font_size=12)
plt.title("Example Graph with 7 Nodes")
plt.show()
```

Custom Graph Class with BFS Implementation

```
import numpy as np

class Graph:
    def __init__(self, num_nodes, edges):
        self.num_nodes = num_nodes
        self.edges = edges
        self.data = [[] for _ in range(num_nodes)]

        # Build adjacency list
        for n1, n2 in edges:
            self.data[n1].append(n2)
            self.data[n2].append(n1)

    def bfs(self, root):
        """
        Breadth-First Search implementation
        Returns: (traversal_order, distances, parents)
        """
        queue = []
        discovered = [False] * len(self.data)
        distance = [None] * len(self.data)
        parent = [None] * len(self.data)

        # Initialize starting node
        distance[root] = 0
        discovered[root] = True
        queue.append(root)
        idx = 0

        # Process queue
        while idx &lt; len(queue):
            current = queue[idx]
            idx += 1

            # Explore neighbors
            for node in self.data[current]:
                if not discovered[node]:
                    distance[node] = 1 + distance[current]
                    parent[node] = current
                    discovered[node] = True
                    queue.append(node)

        return queue, distance, parent

    def check_connected_nodes(self):
        """Check if all nodes are connected"""
        queue, _, _ = self.bfs(self.edges[0][0])
        return len(queue) == self.num_nodes
```

Results Analysis

The BFS algorithm demonstrates:

- **Level-wise exploration**: Nodes are visited in order of their distance from the source
- **Shortest path property**: The distance array contains the minimum number of edges from source to each node
- **Parent tracking**: Enables reconstruction of shortest paths

Example Output:

```
BFS from node 3: [3, 2, 4, 1, 5, 6, 0]
Distances: [2, 2, 1, 0, 1, 2, 2]
Parents: [4, 2, 3, None, 3, 2, 2]
```

Practical 3: Depth-First Search (DFS) Algorithm

Objective

To understand, implement, and analyze the Depth-First Search algorithm for graph traversal and various graph problems.

Theory

What is Depth-First Search?

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack data structure to maintain the traversal order.

Algorithm Characteristics:

- **Time Complexity**: O(V + E) where V is vertices and E is edges
- **Space Complexity**: O(V) for the stack and visited array
- **Memory Usage**: Generally lower than BFS for wide graphs
- **Path Finding**: May not find the shortest path but explores deeply

DFS Algorithm Steps:

1. Start with a source vertex and mark it as visited
2. Add the source vertex to a stack
3. While the stack is not empty:
    - Pop a vertex from the top of the stack
    - If not visited, mark it as visited and add to result
    - For each unvisited neighbor of the current vertex:

- Add the neighbor to the stack

Applications:

- Detecting cycles in graphs
- Topological sorting
- Finding strongly connected components
- Maze solving
- Parsing and syntax analysis
- Backtracking algorithms

DFS vs BFS Comparison:

Aspect	DFS	BFS
Data Structure	Stack	Queue
Memory Usage	Lower for wide graphs	Higher for wide graphs
Path Quality	May not find shortest	Finds shortest path
Completeness	Complete for finite graphs	Always complete
Applications	Cycle detection, topological sort	Shortest path, level traversal

Implementation

```
class Graph:
    def __init__(self, num_nodes, edges):
        self.num_nodes = num_nodes
        self.edges = edges
        self.data = [[] for _ in range(num_nodes)]

        # Build adjacency list
        for n1, n2 in edges:
            self.data[n1].append(n2)
            self.data[n2].append(n1)

    def dfs(self, root):
        """
        Depth-First Search implementation using explicit stack
        Returns: list of nodes in DFS traversal order
        """
        stack = []
        discovered = [False] * len(self.data)
        result = []

        # Start with root node
        stack.append(root)

        while len(stack) &gt; 0:
            current = stack.pop()
```

```
            if not discovered[current]:
                discovered[current] = True
                result.append(current)

                # Add neighbors to stack
                for node in self.data[current]:
                    if not discovered[node]:
                        stack.append(node)

        return result

    def check_connected_nodes(self):
        """Check if all nodes are connected using DFS"""
        result = self.dfs(self.edges[0][0])
        return len(result) == self.num_nodes
```

Algorithm Comparison

Sample Output Analysis:

```
DFS from node 3: [3, 4, 0, 1, 2, 6, 5]
BFS from node 3: [3, 2, 4, 1, 5, 6, 0]
```

The difference in traversal order demonstrates:

- **DFS**: Follows one path deeply before exploring alternatives
- **BFS**: Explores all neighbors at current level before going deeper

Conclusion

This practical guide has covered three fundamental areas of AI and computer science:

Key Takeaways:

1. **Python Libraries for AI**: Understanding the ecosystem is crucial for efficient AI development
2. **BFS Algorithm**: Optimal for shortest path problems and level-wise exploration
3. **DFS Algorithm**: Efficient for deep exploration and cycle detection

Applications in Real World:

- **AI Development**: Using appropriate libraries accelerates development
- **Graph Algorithms**: Essential for network analysis, pathfinding, and decision trees
- **Problem Solving**: Both algorithms serve different purposes based on problem requirements

Skills Developed:

- Python programming for AI applications
- Algorithm implementation and analysis
- Graph theory and data structure manipulation
- Performance comparison and optimization

Next Steps:

- Explore advanced graph algorithms (Dijkstra's, A*)
- Implement machine learning models using covered libraries
- Apply algorithms to real-world problems
- Study algorithm complexity and optimization techniques

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
2. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson.
3. NetworkX Documentation: https://networkx.org/
4. Python Software Foundation: https://www.python.org/
5. Scikit-learn Documentation: https://scikit-learn.org/
6. TensorFlow Documentation: https://www.tensorflow.org/

Document prepared for academic purposes
Campus AI Practicals Guide
Version 1.0 - 2025