AI Practicals

**A Comprehensive Guide to Artificial Intelligence Algorithms and Implementations**

Table of Contents

1. [Introduction](#introduction)
2. [Practical 1: Python Libraries for AI Development](#practical-1-python-libraries-for-ai-development)
3. [Practical 2: Breadth-First Search (BFS) Algorithm](#practical-2-breadth-first-search-bfs-algorithm)
4. [Practical 3: Depth-First Search (DFS) Algorithm](#practical-3-depth-first-search-dfs-algorithm)

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)  
  
# LangGraph Example (requires installation)  
import langgraph  
graph = langgraph.Graph()  
graph.add\_node("start")  
print(graph.nodes)  
  
# 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))  
  
# OpenCV Example  
import cv2  
import numpy as np  
img = np.zeros((100,100,3), np.uint8)  
cv2.circle(img, (50,50), 30, (255,0,0), -1)  
cv2.imwrite('circle.png', img)  
  
# Streamlit Example (run with: streamlit run <filename>)  
import streamlit as st  
st.write("Hello, Streamlit!")  
  
# FastAPI Example (run with: uvicorn <filename>:app)  
from fastapi import FastAPI  
app = FastAPI()  
@app.get("/")  
def read\_root():  
 return {"Hello": "FastAPI"}  
  
# Django Example (run with: python manage.py runserver)  
from django.http import HttpResponse  
def hello(request):  
 return HttpResponse("Hello, Django!")

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, font\_family='sans-serif')  
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 \_\_repr\_\_(self):  
 return "\n".join(["{} : {}".format(n, x) for n, x in enumerate(self.data)])  
  
 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 < 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

Example Usage

# Create and test the graph  
g1 = Graph(num\_nodes, edges)  
print("Graph representation:")  
print(g1)  
  
# Add an edge and test  
g1.add\_edge(2, 6)  
print("\nAfter adding edge (2,6):")  
print(g1)  
  
# Perform BFS from node 3  
traversal, distances, parents = g1.bfs(3)  
print(f"\nBFS from node 3:")  
print(f"Traversal order: {traversal}")  
print(f"Distances: {distances}")  
print(f"Parents: {parents}")  
  
# Check connectivity  
print(f"\nIs graph connected? {g1.check\_connected\_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

Learning Outcomes

* Understanding of BFS algorithm and its applications
* Implementation of queue-based graph traversal
* Analysis of algorithm complexity and optimality

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 (often implemented recursively using the call stack) 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

Graph Setup (Same as BFS)

import networkx as nx  
import matplotlib.pyplot as plt  
  
# Create the same graph structure  
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, font\_family='sans-serif')  
plt.title("Example Graph with 7 Nodes")  
plt.show()

Custom Graph Class with DFS 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 \_\_repr\_\_(self):  
 return "\n".join(["{} : {}".format(n, x) for n, x in enumerate(self.data)])  
  
 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) > 0:  
 current = stack.pop()  
  
 if not discovered[current]:  
 discovered[current] = True  
 result.append(current)  
  
 # Add neighbors to stack (in reverse order for consistent traversal)  
 for node in self.data[current]:  
 if not discovered[node]:  
 stack.append(node)  
  
 return result  
  
 def dfs\_recursive(self, root, discovered=None, result=None):  
 """  
 Recursive DFS implementation  
 """  
 if discovered is None:  
 discovered = [False] \* len(self.data)  
 if result is None:  
 result = []  
  
 discovered[root] = True  
 result.append(root)  
  
 for neighbor in self.data[root]:  
 if not discovered[neighbor]:  
 self.dfs\_recursive(neighbor, discovered, result)  
  
 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  
  
 def has\_cycle(self):  
 """Detect if graph has a cycle using DFS"""  
 visited = [False] \* self.num\_nodes  
 rec\_stack = [False] \* self.num\_nodes  
  
 def dfs\_cycle(node):  
 visited[node] = True  
 rec\_stack[node] = True  
  
 for neighbor in self.data[node]:  
 if not visited[neighbor]:  
 if dfs\_cycle(neighbor):  
 return True  
 elif rec\_stack[neighbor]:  
 return True  
  
 rec\_stack[node] = False  
 return False  
  
 for node in range(self.num\_nodes):  
 if not visited[node]:  
 if dfs\_cycle(node):  
 return True  
 return False

Example Usage

# Create and test the graph  
g1 = Graph(num\_nodes, edges)  
print("Graph representation:")  
print(g1)  
  
# Add an edge and test  
g1.add\_edge(2, 6)  
print("\nAfter adding edge (2,6):")  
print(g1)  
  
# Perform DFS from node 3  
dfs\_result = g1.dfs(3)  
print(f"\nDFS from node 3 (iterative): {dfs\_result}")  
  
# Perform recursive DFS  
dfs\_recursive\_result = g1.dfs\_recursive(3)  
print(f"DFS from node 3 (recursive): {dfs\_recursive\_result}")  
  
# Check connectivity  
print(f"\nIs graph connected? {g1.check\_connected\_nodes()}")  
  
# Check for cycles  
print(f"Does graph have cycles? {g1.has\_cycle()}")

Algorithm Comparison

DFS Traversal Characteristics:

* **Deep Exploration**: Goes as deep as possible before backtracking
* **Stack-based**: Uses LIFO (Last In, First Out) principle
* **Memory Efficient**: Better for graphs with high branching factor
* **Non-optimal**: May not find shortest paths

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

Learning Outcomes

* Understanding of DFS algorithm and stack-based traversal
* Implementation of both iterative and recursive approaches
* Application to cycle detection and connectivity problems
* Comparison with BFS algorithm