EX.NO:	FINDING SHORTEST PATH USING GRAPHS
DATE: / /2025	

AIM

To Implement finding shortest path using graphs in python.

SOURCE CODE:

```
import csv
import heapq
import sys
from typing import Dict, List, Tuple, Union
Weight = Union[int, float]
Graph = Dict[str, List[Tuple[str, Weight]]]
def load_graph(csv_file: str, undirected: bool = True) -> Graph:
    """Load graph from CSV file into adjacency list.
    Each row must have source,target,weight """
    graph: Graph = {}
    try:
        with open(csv_file, newline='', encoding='utf-8') as f:
            reader = csv.DictReader(f)
            if not {"source", "target", "weight"}.issubset(reader.fieldnames or []):
                raise ValueError("CSV must contain headers: source,target,weight")
            for row in reader:
                src = row["source"].strip()
                tgt = row["target"].strip()
                try:
                    weight = float(row["weight"])
                except Exception:
                    raise ValueError(f"Invalid weight value on row: {row}")
                graph.setdefault(src, [])
                graph.setdefault(tgt, [])
                graph[src].append((tgt, weight))
                if undirected:
                    graph[tgt].append((src, weight))
    except FileNotFoundError:
        raise FileNotFoundError(f"CSV file not found: {csv_file}")
    return graph

def dijkstra(graph: Graph, start: str, end: str) -> Tuple[Union[List[str], None],
float]:
    """Return (path_list, total_distance). If no path: (None, inf)."""
    if start not in graph or end not in graph:
        return None, float("inf")
    if start == end:
        return [start], 0.0
    pq: List[Tuple[float, str]] = [(0.0, start)]
    distances: Dict[str, float] = {node: float("inf") for node in graph}
    distances[start] = 0.0
    parent: Dict[str, Union[str, None]] = {node: None for node in graph}
    visited: Dict[str, bool] = {}
    while pq:
        curr_dist, curr_node = heapq.heappop(pq)
        if visited.get(curr_node, False):
            continue
        visited[curr_node] = True
        if curr_node == end:
            break
        for neighbor, w in graph.get(curr_node, []):
            if visited.get(neighbor, False):
                continue
            new_dist = curr_dist + w
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                parent[neighbor] = curr_node
                heapq.heappush(pq, (new_dist, neighbor))
    if distances[end] == float("inf"):
        return None, float("inf")
    path: List[str] = []
    node = end
    while node is not None:
        path.append(node)
        node = parent[node]
    path.reverse()
    return path, distances[end]
def pretty_print_graph(graph: Graph) -> None:
    print("Graph adjacency list (node: [(neighbor, weight), ...])")
    for node in sorted(graph.keys()):
        print(f"  {node}: {graph[node]}")
    print()
def main():
    csv_file = "graph_dataset.csv"
    try:
        graph = load_graph(csv_file, undirected=True)
    except Exception as e:
        print("Error loading graph:", e)
        sys.exit(1)
    print("Graph loaded successfully from", csv_file)
    pretty_print_graph(graph)
    while True:
        try:
            start = input("Enter START node (or 'exit' to quit): ").strip()
        except (EOFError, KeyboardInterrupt):
            print("\nExiting.")
            break
        if start.lower() == "exit":
            print("Goodbye.")
            break
        end = input("Enter END node: ").strip()
        if start not in graph:
            print(f"Start node '{start}' not found in graph. Available nodes: {', 
'.join(sorted(graph.keys()))}\n")
            continue
        if end not in graph:
            print(f"End node '{end}' not found in graph. Available nodes: {', 
'.join(sorted(graph.keys()))}\n")
            continue
        path, dist = dijkstra(graph, start, end)
        if path is None:
            print(f"No path found from {start} to {end}.\n")
        else:
            if dist == int(dist):
                dist_str = str(int(dist))
            else:
                dist_str = f"{dist:.4f}"
            print(f"Shortest Path: {' → '.join(path)}")
            print(f"Total Distance: {dist_str}\n")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Graph loaded successfully from graph_dataset.csv
Graph adjacency list (node: [(neighbor, weight), ...])
  A: [('B', 4.0), ('C', 2.0)]
  B: [('A', 4.0), ('C', 5.0), ('D', 10.0)]
  C: [('A', 2.0), ('B', 5.0), ('E', 3.0)]
  D: [('B', 10.0), ('E', 4.0), ('F', 11.0)]
  E: [('C', 3.0), ('D', 4.0), ('F', 2.0)]
  F: [('D', 11.0), ('E', 2.0)]

Enter START node (or 'exit' to quit): A
Enter END node: F

Shortest Path: A → C → E → F
Total Distance: 7

Enter START node (or 'exit' to quit): A
Enter END node: D

Shortest Path: A → C → E → D
Total Distance: 9

Enter START node (or 'exit' to quit): C
Enter END node: B

Shortest Path: C → A → B
Total Distance: 6

Enter START node (or 'exit' to quit): Z
Start node 'Z' not found in graph. Available nodes: A, B, C, D, E, F

Enter START node (or 'exit' to quit): exit
Goodbye.
```

RESULT:

The program has been successfully executed.