

LAB 4

GRAPHS AND TREES

Name	ID
Ranime Ahmed Elsayed Shehata.	21010531
Youssef Tarek Hussein Yousry.	21011595

Problem (1): Airline Network Shortest-Path

Finder:

❖ Problem Statement:

Imagine you are tasked with developing a tool to assist airline passengers in finding the most efficient route between two airports within the airline's network. Create a Java program that models an airline network as a graph, where airports are nodes and flights are edges. Your program should enable users to input details of the flight connections between airports and find the shortest path between a specified source and destination airport.

- Implement a class to represent the airline network graph. Each airport is a node, and flights between airports are edges. You can choose an adjacency matrix or adjacency list to represent the graph.
- Implement Dijkstra's algorithm to find the shortest path between two specified airports in the airline network.
- Display the optimal route details, including the sequence of airports to visit and the total distance or time required for the journey.
- Implement error handling mechanisms to handle cases where the specified source or destination airport is not in the network or when there is no direct flight between them.

Example Input

Enter the list of airports: A, B, C, D, E

Enter the flights: A-B, B-C, C-D, D-E, A-C, B-D

The distance for each flight (in miles)

A-B: 500

B-C: 300

C-D: 400

D-E: 600

A-C: 700

B-D: 450

Enter source airport: A

Enter destination airport: E

Example Output Shortest path from A to E: A - B - D - E

Total distance: 1550 miles

❖ Code Explanation:

In this code two classes, 2 classes are defined: **AirportGraph** and **AirlineRouteFinder**, to represent and find the shortest path in a graph of airports connected by flights. The program uses **Dijkstra's algorithm** to find the shortest path between a given source and destination airport.

AirportGraph class:

- **graph**: A map representing the graph of airports and flights. The outer map (``graph``) has airport codes as keys, and each value is another map (`Map<String, Integer>`) representing the neighboring airports and their distances.
- **AirportGraph**: It's a Constructor that takes three parameters: ``airports`` (List of airport codes), ``flights`` (List of flight codes), and ``distances`` (Map of flight codes to distances in miles). Initializes the ``graph`` by creating nodes for each airport and edges for each flight, setting distances accordingly.
- **findShortestPath**: It takes ``source`` and ``destination`` as parameters and returns the shortest path from the source to the destination using Dijkstra's algorithm. Initializes data structures for tracking distances, previous nodes, visited nodes, and uses a priority queue for efficient node exploration. Implements Dijkstra's algorithm to compute the shortest path and reconstructs and returns the path from source to destination.

AirlineRouteFinder class:

main:

- Takes user input for airports, flights, and distances.
- Creates an instance of "AirportGraph" using the provided data.
- Takes user input for the source and destination airports.
- Validates user inputs for source and destination airports.
- Finds the shortest path and total distance using the "findShortestPath" method.
- Displays the result, including the shortest path and total distance.

calculateTotalDistance:

- Takes a path (List of airports) and a map of distances.

- Calculates the total distance by summing the distances of consecutive flights in the given path.

Input Validation:

- The code includes input validation to ensure that distances are non-negative and that the source and destination airports are valid.
- Airport and flight codes are stored and compared in uppercase to ensure case-insensitive matching.
- The program exits with an error message if there are issues with input validation.

Flow:

1. User inputs the list of airports, flights, and distances.
2. The `AirportGraph` is constructed using the input data.
3. User inputs the source and destination airports with validation.
4. Dijkstra's algorithm is used to find the shortest path.
5. The result, including the shortest path and total distance, is displayed.

❖ Used data structures:

The data structures used are HashMaps for representing the graph and distances, HashSet for tracking visited airports, PriorityQueue for efficient node selection in Dijkstra's algorithm, and ArrayList for storing lists of airports, flights, and the final shortest path.

1. HashMap (graph):

- Used to represent the graph of airports and flights.
- The outer map (graph) has airport codes as keys.
- The inner map (Map<String, Integer>) has neighboring airport codes as keys and the corresponding distances as values.

2. HashMap (distances):

- Used to store the minimum distances from the source airport to all other airports during the execution of Dijkstra's algorithm.

3. HashMap (previous):

- Used to store the previous airport in the shortest path from the source to each airport.

4. HashSet (visited):

- Used to keep track of visited airports during the algorithm to avoid redundant processing.

5. PriorityQueue (priorityQueue):

- Used to efficiently select the next airport to explore based on the current minimum distance.

6. ArrayList (airports, flights, shortestPath):

- Used to store the list of airports, flights, and the final shortest path.

7. HashMap (distances):

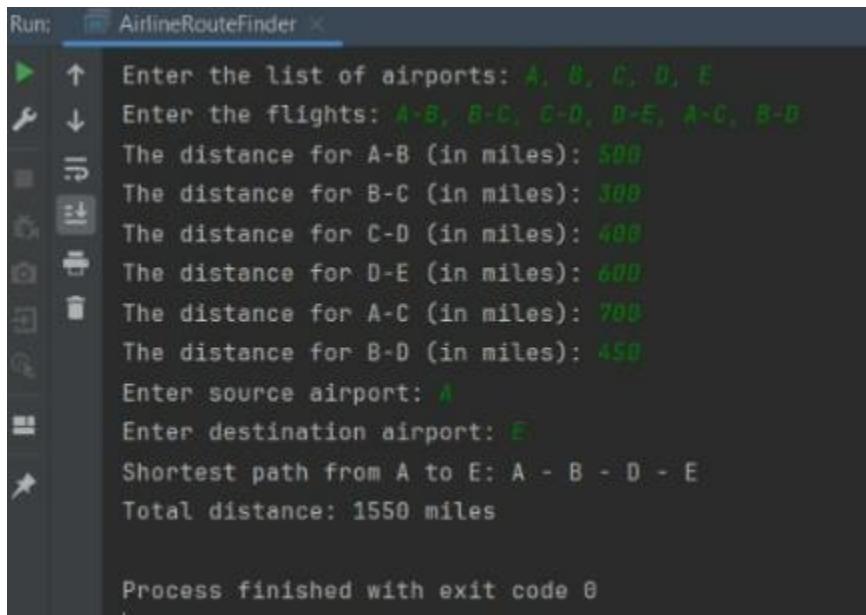
- Used to store the distances for each flight.

8. HashMap (distances):

- Used to look up distances for each flight in the path.

❖ Sample runs and different test case:

- The example given in the lab:



```
Run: AirlineRouteFinder
Enter the list of airports: A, B, C, D, E
Enter the flights: A-B, B-C, C-D, D-E, A-C, B-D
The distance for A-B (in miles): 500
The distance for B-C (in miles): 300
The distance for C-D (in miles): 400
The distance for D-E (in miles): 600
The distance for A-C (in miles): 700
The distance for B-D (in miles): 450
Enter source airport: A
Enter destination airport: E
Shortest path from A to E: A - B - D - E
Total distance: 1550 miles

Process finished with exit code 0
```

- When entering an invalid destination or source:

```
Run: AirlineRouteFinder x
"C:\Users\Kimo Store\.jdk\openjdk-21.0.1\bin\java.exe"
Enter the list of airports: A, B, C, D, E
Enter the flights: A-B, B-C, C-D, D-E, A-C, B-D
The distance for A-B (in miles): 500
The distance for B-C (in miles): 300
The distance for C-D (in miles): 400
The distance for D-E (in miles): 600
The distance for A-C (in miles): 700
The distance for B-D (in miles): 450
Enter source airport: A
Enter destination airport: X
Destination airport not found. Exiting.

Process finished with exit code 1
```

- When there's no direct flights:

```
Run: AirlineRouteFinder x
"C:\Users\Kimo Store\.jdk\openjdk-21.0.1\bin\java.exe"
Enter the list of airports: A, B, C, D
Enter the flights: A-B, B-C, A-C
The distance for A-B (in miles): 5
The distance for B-C (in miles): 3
The distance for A-C (in miles): 4
Enter source airport: A
Enter destination airport: D
No flights available from A to D

Process finished with exit code 1
```

- Handling the upper and lower case inputs:

```
Run: AirlineRouteFinder x
"C:\Users\Kimo Store\.jdk\openjdk-21.0.1\bin\java.exe"
Enter the list of airports: A, B, c, d, e
Enter the flights: A-B, B-C, c-d, d-e, A-C, B-D
The distance for A-B (in miles): 500
The distance for B-C (in miles): 300
The distance for C-D (in miles): 400
The distance for D-E (in miles): 600
The distance for A-C (in miles): 700
The distance for B-D (in miles): 450
Enter source airport: a
Enter destination airport: e
Shortest path from A to E: A - B - D - E
Total distance: 1550 miles

Process finished with exit code 0
```

Problem (2): Class Schedule Optimization:

❖ Problem Statement:

Imagine a school with multiple classes and subject timings where certain classes cannot occur simultaneously due to shared resources or teacher availability. Develop a Java program that generates an optimized class schedule by assigning time slots to classes, ensuring that no conflicting classes occur at the same time.

- Represent the schedule information as a graph where nodes represent classes, and edges denote conflicting timings between classes.
- Implement a graph coloring algorithm to assign distinct colors (time slots) to nodes (classes) in the graph. Ensure that adjacent nodes (classes) linked by edges (conflicting timings) do not share the same color (time slot) to avoid scheduling conflicts.
- Display the timetable with color-coded class timings, ensuring that conflicting classes have different colors (non-overlapping timings).
- Use any color names as you like.

Example Input

Classes: A, B, C, D

Conflicting classes (cannot occur simultaneously): A-B B-C

Example Output

Optimized Class Schedule: A - Blue B - Red C - Blue D – Green

❖ Code Explanation:

This code is an implementation of a graph coloring algorithm using a greedy approach. The goal is to assign colors to vertices of a graph such that no two adjacent vertices have the same color, satisfying a set of conflicts.

1. adjacencyFinder method:

- Parameters:
 - input: An array of characters representing vertices.
 - adjacency: A 2D array to store the adjacency matrix of the graph.
 - conflicts: An array of strings representing conflicts between vertices.
 - num: The number of conflicts.

- This method finds the indices of conflicting vertices in the input array and updates the adjacency matrix accordingly. If vertices 'a' and 'b' are in conflict, it sets `adjacency[a][b] = 1` and `adjacency[b][a] = 1`.

2. applyGreedy method:

- Parameters:
 - colored: An array to store the color assigned to each vertex.
 - adjacency: The adjacency matrix of the graph.
- This method applies a greedy coloring algorithm. It iterates through each vertex and assigns the smallest available color to it, ensuring that no adjacent vertices have the same color.

3. main:

- It is the entry point of the program.
- It takes input from the user in the form of comma-separated vertices and the number of conflicts.
- It initializes an array of colors and sets up data structures to represent the graph and conflicts.
- It then calls (adjacencyFinder) to update the adjacency matrix based on conflicts.
- It initializes an array to store the color of each vertex and calls (applyGreedy) to perform the greedy coloring.
- Finally, it prints the vertices along with their assigned colors.

4. User Input:

- The user is prompted to enter comma-separated vertices, the number of conflicts, and each conflict in the form 'a-b'.

5. Output:

- The program outputs the vertices along with their assigned colors after applying the greedy coloring algorithm.

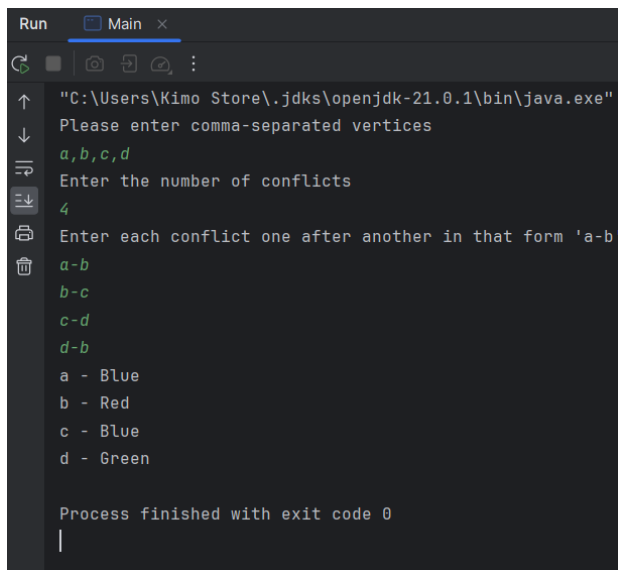
❖ Used data structures:

Arrays:

- char[] inputarray: An array to store the vertices of the graph.
- int[][] adjacency: A 2D array to represent the adjacency matrix of the graph.
- String[] conflicts: An array to store conflict strings entered by the user.
- String[] colors: An array of strings representing different colors.
- int[] colored: An array to store the color assigned to each vertex.

❖ Sample runs and different test cases:

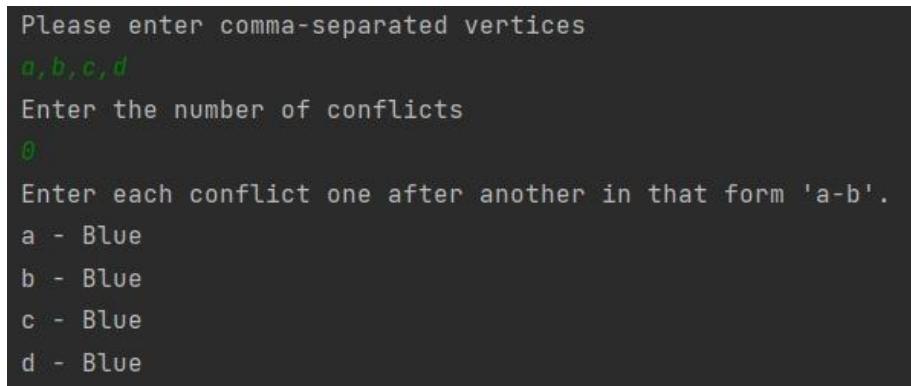
• The given example in the lab:



```
Run Main x
"C:\Users\Kimo Store\.jdk\openjdk-21.0.1\bin\java.exe"
Please enter comma-separated vertices
a,b,c,d
Enter the number of conflicts
4
Enter each conflict one after another in that form 'a-b'
a-b
b-c
c-d
d-b
a - Blue
b - Red
c - Blue
d - Green

Process finished with exit code 0
```

• No conflicts:



```
Please enter comma-separated vertices
a,b,c,d
Enter the number of conflicts
0
Enter each conflict one after another in that form 'a-b'.
a - Blue
b - Blue
c - Blue
d - Blue
```

- Extra runs:

```
a,b,c,d
Enter the number of conflicts
2
Enter each conflict one after another in that form 'a-b'.
a-c
b-d
a - Blue
b - Blue
c - Red
d - Red
```

```
Please enter comma-separated vertices
a,b,c,d
Enter the number of conflicts
6
Enter each conflict one after another in that form 'a-b'.
a-b
a-c
a-d
b-c
b-d
c-d
a - Blue
b - Red
c - Green
d - Yellow
```

Please enter comma-separated vertices

a,b,c,d,e,f,g

Enter the number of conflicts

12

Enter each conflict one after another in that form 'a-b'.

a-b

a-c

b-e

b-c

b-d

c-d

c-f

d-e

d-f

e-f

e-g

f-g

a - Blue

b - Red

c - Green

d - Blue

e - Green

f - Red

g - Blue

Problem (3): Tree Traversal:

❖ Problem Statement:

Implement three algorithms for Binary Tree traversal recursively or iteratively

- Preorder
- Inorder
- Postorder

Example input

Enter the value of the root node: 1

Enter left child value of 1 (or -1 to skip): 2

Enter right child value of 1 (or -1 to skip): 3

Enter left child value of 2 (or -1 to skip): 4

Enter right child value of 2 (or -1 to skip): 5

Enter left child value of 3 (or -1 to skip): -1

Enter right child value of 3 (or -1 to skip): 6

Enter left child value of 4 (or -1 to skip): -1

Enter right child value of 4 (or -1 to skip): -1

Enter left child value of 5 (or -1 to skip): -1

Enter right child value of 5 (or -1 to skip): -1

Enter left child value of 6 (or -1 to skip): -1

Enter right child value of 6 (or -1 to skip): -1

Example Output

Preorder Traversal: 1 2 4 5 3 6

Inorder Traversal: 4 2 5 1 3 6

Postorder Traversal: 4 5 2 6 3 1

❖ Code Explanation:

This code implements binary tree traversal algorithms using both recursive and iterative approaches. The program allows the user to build a binary tree and choose between recursive and iterative methods for performing pre-order, in-order, and post-order traversal.

1. TreeNode Class:

Defines a simple `TreeNode` class for the binary tree nodes, with an integer value (`val`) and references to the left and right children.

2. buildTree Method:

Recursively builds a binary tree by taking user input for each node's value and its left and right children. The user can enter `-1` to skip adding a child.

3. Recursive Traversal Methods:

- `preorderRecursive`, `inorderRecursive`, and `postorderRecursive`: These methods perform pre-order, in-order, and post-order traversals, respectively, using recursive algorithms. They are the entry point for users and sets up the initial state (e.g., creating a `StringBuilder` for storing traversal results).

4. Recursive Helper Methods:

- `preorderRecursiveHelper`, `inorderRecursiveHelper`, and `postorderRecursiveHelper`: These helper methods are used by the corresponding recursive traversal methods to perform the actual traversal. They do the actual recursive work of traversing the tree and updating the result accordingly.

5. Iterative Traversal Methods:

- `preorderIterative`, `inorderIterative`, and `postorderIterative`: These methods perform pre-order, in-order, and post-order traversals, respectively, using iterative algorithms.

6. Main:

Implements a simple console-based menu where the user can choose between recursive and iterative traversal methods, build a binary tree, and see the results of the chosen traversal.

- The program runs in an infinite loop until the user chooses to exit.
- The user can choose between recursive and iterative methods for traversing the binary tree.
- The program exits if the user chooses option `3`.

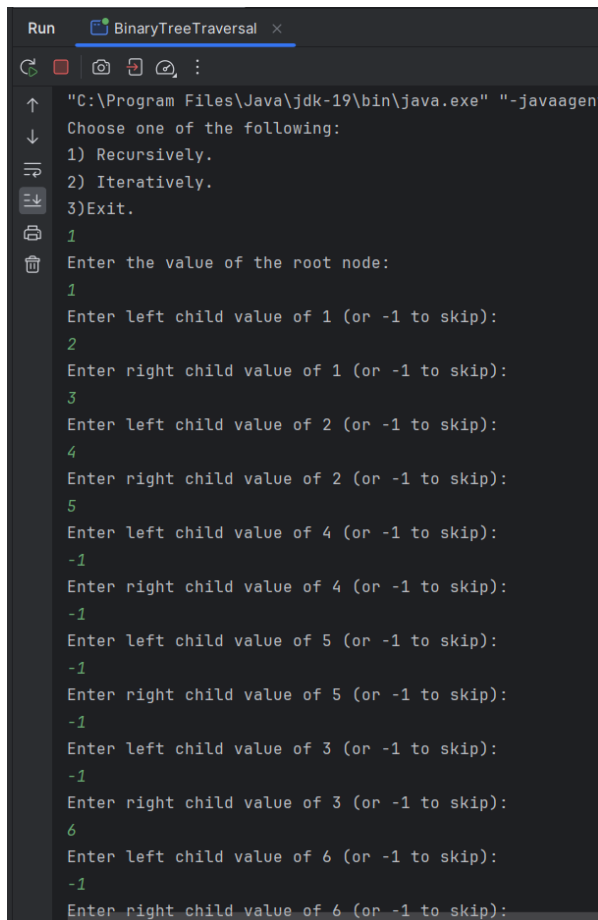
❖ Used data structures:

Stack:

- **Stack<TreeNode>**: used to store the nodes during the traversal. Nodes are popped from the stack, and their values are appended to the result in the pre-order sequence (`node - left - right`).

❖ Sample runs and different test cases:

- The example given in the lab using both iterative and recursive approaches:



```
Run BinaryTreeTraversal x
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
1
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
2
Enter right child value of 1 (or -1 to skip):
3
Enter left child value of 2 (or -1 to skip):
4
Enter right child value of 2 (or -1 to skip):
5
Enter left child value of 4 (or -1 to skip):
-1
Enter right child value of 4 (or -1 to skip):
-1
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 3 (or -1 to skip):
-1
Enter right child value of 3 (or -1 to skip):
6
Enter left child value of 6 (or -1 to skip):
-1
Enter right child value of 6 (or -1 to skip):
```

```
Run BinaryTreeTraversal x
Enter right child value of 6 (or -1 to skip):
-1
Preorder Traversal: 1 2 4 5 3 6
Inorder Traversal: 4 2 5 1 3 6
Postorder Traversal: 4 5 2 6 3 1
-----
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
2
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
2
Enter right child value of 1 (or -1 to skip):
3
Enter left child value of 2 (or -1 to skip):
4
Enter right child value of 2 (or -1 to skip):
5
Enter left child value of 4 (or -1 to skip):
-1
Enter right child value of 4 (or -1 to skip):
-1
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 3 (or -1 to skip):
-1
```

```
Enter left child value of 3 (or -1 to skip):
-1
Enter right child value of 3 (or -1 to skip):
6
Enter left child value of 6 (or -1 to skip):
-1
Enter right child value of 6 (or -1 to skip):
-1
Preorder Traversal: 1 2 4 5 3 6
Inorder Traversal: 4 2 5 1 3 6
Postorder Traversal: 4 5 2 6 3 1
-----
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
3
Exiting the program.
```

- Extra runs (iterative and recursive):

```
Run BinaryTreeTraversal x
"\"C:\\Program Files\\Java\\jdk-19\\bin\\java.exe\" \"-javaa
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
1
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
2
Enter right child value of 1 (or -1 to skip):
3
Enter left child value of 2 (or -1 to skip):
4
Enter right child value of 2 (or -1 to skip):
5
Enter left child value of 4 (or -1 to skip):
-1
Enter right child value of 4 (or -1 to skip):
-1
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 3 (or -1 to skip):
-1
Enter right child value of 3 (or -1 to skip):
-1
Preorder Traversal: 1 2 4 5 3
Inorder Traversal: 4 2 5 1 3
Postorder Traversal: 4 5 2 3 1
```

```
Run BinaryTreeTraversal x
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
2
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
2
Enter right child value of 1 (or -1 to skip):
3
Enter left child value of 2 (or -1 to skip):
4
Enter right child value of 2 (or -1 to skip):
5
Enter left child value of 4 (or -1 to skip):
-1
Enter right child value of 4 (or -1 to skip):
-1
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 3 (or -1 to skip):
-1
Enter right child value of 3 (or -1 to skip):
-1
Preorder Traversal: 1 2 4 5 3
Inorder Traversal: 4 2 5 1 3
Postorder Traversal: 4 5 2 3 1
-----
```


- Empty tree:

```
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
1
Enter the value of the root node:
-1
Preorder Traversal:
Inorder Traversal:
Postorder Traversal:
```

- Single node tree:

```
Choose one of the following:
1) Recursively.
2) Iteratively.
3)Exit.
1
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
-1
Enter right child value of 1 (or -1 to skip):
-1
Preorder Traversal: 1
Inorder Traversal: 1
Postorder Traversal: 1
```