Java Assignment: Breadth First Search Algorithm Implementation and Graph Modeling

**You may use the JCF**for this assignment.

Your task is to implement the Breadth First Search (BFS) algorithm for graphs. You can refer to the pseudocode provided in the book on page 603 (provided below), under the section 'IsPathBF', as a reference for writing your code.

Utilize the data structure (an array with link lists off of each array cell), so that we can keep track of each node, and all vertices it is adjacent to.

A diagram of a diagram

Description automatically generated with medium confidence

Next, create a graph modeling a domain of your interest. This could encompass various domains such as flight routes, social relationships, commute routes, or even a shopping spree. Aim for creativity and select a domain that captivates your interest. Your model should comprise a minimum of 6 nodes, with each node having at least three connections to other nodes. Ensure that your code is designed to be generic.

Output:

Provide the Adjacency Table (or List) of your model to showcase the structure of your graph.

Demonstrate three distinct paths through your graph. For instance, if you're modeling flight routes, you might illustrate paths from:  
a) New York to Jerusalem  
b) Paris to Rome  
c) Tel Aviv to Paris

**Extra Credit**: Implement Dijkstra's algorithm to determine the shortest path within your graph. You can opt to prioritize either the greatest or least priority, depending on your preference and the application of your model.

Let’s start:

1. Breadth First Search Implementation  
   - Correct implementation of BFS algorithm  
   - Proper handling of edge cases and error checking  
   - Efficient data structures and algorithms usage  
   - Clear and concise code readability

2. Graph Modeling  
   - Creation of a domain-specific graph with a minimum of 6 nodes  
   - Each node has at least three connections to other nodes  
   - Generic design allowing flexibility in graph representation

3. Output Presentation   
   - Accurate printing of Adjacency Table or List  
   - Clear demonstration of three different paths through the graph  
   - Well-organized and easy-to-follow output

4. Object-Oriented Design  
   - Clear identification of classes and their responsibilities  
   - Effective use of abstraction and encapsulation principles  
   - Proper inheritance and polymorphism (if applicable)  
   - Documentation and comments reflecting design decisions

5. Overall Code Quality  
   - Consistency in coding style and conventions  
   - Proper documentation and comments  
   - Modularity and reusability of code components  
   - Error handling and robustness

Extra Credit - Dijkstra's Algorithm Implementation   
   - Correct implementation of Dijkstra's algorithm  
   - Consideration of priority choice and application relevance   
   - Efficient utilization of data structures and algorithms

pseudocode provided in the book on page 603:

|  |
| --- |
| **IsPathBF (startVertex, endVertex): returns boolean** |
| Set found to false Clear all marks Mark the startVertex Enqueue the startVertex into the queue  do  Set current vertex = queue.dequeue()  if current vertex equals endVertex  Set found to true  else  for each adjacent vertex  if adjacent vertex is not marked  Mark the adjacent vertex and  Enqueue it into the queue  while !queue.isEmpty() AND !found  return found |