Homework 4: Problem 3 - Graph Representation

1 Graph Representation

For the Graph ADT implementation, I chose to use an **adjacency list** representation with the following structure:

```
1 private Map<String, Map<String, List<String>>> nodeEdges;
```

This representation uses:

- A primary map where keys are node data (String) representing the nodes in the graph
- For each node, a nested map where keys are destination node data (String) and values are lists of edge labels (List¡String¿) representing edges from the source node to the destination node

2 Advantages of Different Representations

2.1 1. Collection of Edges

- **Advantage**: Simplicity of implementation; easy to add and remove edges.
- **Disadvantage**: Inefficient for finding all outgoing or incoming edges for a specific node, requiring O(—E—) time where —E— is the number of edges.

2.2 2. Adjacency List (My Choice)

- **Advantage**: Efficient for sparse graphs (where —E— ¡¡ —V—^{2}); provides O(1) access to a node's outgoing edges.
- **Advantage**: Memory-efficient for sparse graphs, using space proportional to —V— + —E—.
- **Advantage**: Fast iteration over a node's neighbors (constant time per neighbor).
- **Disadvantage**: Not as efficient as adjacency matrix for checking if an edge exists between two specific nodes.

2.3 3. Adjacency Matrix

- **Advantage**: Very fast (O(1)) edge existence checks between any two nodes.
- **Advantage**: Simple to implement for basic graph operations.
- **Disadvantage**: Memory-inefficient for sparse graphs, always requiring O(—V—^{2}) space.
- **Disadvantage**: Slower for operations that need to iterate over all edges or neighbors of a node.

3 Rationale for my Choice

I chose the adjacency list representation for the following reasons:

1. **Efficiency for Common Operations**: Most graph algorithms need to quickly find all neighbors (children) of a given node, which is very efficient with adjacency lists.
2. **Support for Multigraph**: my representation easily accommodates multiple edges between the same pair of nodes by storing a list of edge labels for each destination node.
3. **Space Efficiency**: Real-world graphs tend to be sparse (having far fewer edges than the theoretical maximum), making adjacency lists much more space-efficient than adjacency matrices.
4. **Flexibility**: The nested map structure allows us to efficiently implement additional graph operations, such as getting all edge labels between two nodes or checking the existence of specific edges.
5. **Scalability**: As mentioned in the assignment, the path-finding application will create and operate on very large graphs. The adjacency list representation scales better with graph size compared to an adjacency matrix, especially for sparse graphs.

My implementation ensures that all operations required by the GraphWrapper interface (listNodes, listChildren, and listChildrenXML) can be performed efficiently. By using HashMaps for the outer and inner maps, we achieve O(1) average time complexity for node and edge lookups, which is critical for good performance in large graphs.