Yes, the DFS (Depth-First Search) logic in your `Graph` class is correct. Here's why:

Key Points of Your DFS Implementation:
1. **Initialization**:
   - You initialize a `visited` array to keep track of visited nodes (all set to `false` initially).
   - You start the traversal from the `sourceNode`.

2. **Recursive DFS Helper (`dfsTraversalHelper`)**:
   - Marks the current node as `visited`.
   - Adds the current node to the `result` list.
   - Recursively visits all unvisited neighbors.

3. **Handling Neighbors**:
   - For each neighbor of the current node, if it hasn't been visited, the DFS continues recursively.

Why It Works:
- Your implementation follows the standard DFS approach:
  1. Visit a node.
  2. Mark it as visited.
  3. Recursively visit all its unvisited neighbors.
- The `visited` array ensures no node is processed more than once, preventing infinite loops in cycles.
- The adjacency list correctly represents the graph, and the traversal covers all reachable nodes from the `sourceNode`.

Minor Suggestions (Optional Improvements):
1. **Input Validation**:
   - Check if `sourceNode` is valid (i.e., `0 <= sourceNode < numberOfNodes`). Otherwise, throw an exception.
   - Example:
     ```java
     if (sourceNode < 0 || sourceNode >= numberOfNodes) {
         throw new IllegalArgumentException("Invalid source node");
     }
     ```

2. **Immutable Results**:
   - Return an immutable list (e.g., `List.copyOf(result)`) to prevent external modifications.
   - Example:
     ```java
     return List.copyOf(result);
     ```

3. **Handling Disconnected Graphs**:
   - If you want to traverse all nodes (even in disconnected graphs), you can modify `traverseDFS` to iterate over all nodes and call `dfsTraversalHelper` on unvisited nodes.

Example Usage:
Here’s how you might use your `Graph` class:
```java
public static void main(String[] args) {
    Graph graph = new Graph(5); // 5 nodes (0 to 4)
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    List<Integer> dfsResult = graph.traverseDFS(0);
    System.out.println(dfsResult); // Output depends on edge order, e.g., [0, 1, 3, 2, 4]
}
```

Final Answer:
Your DFS implementation is **correct** for traversing all reachable nodes from a given `sourceNode` in an undirected graph. The logic follows the standard DFS approach, and the code is well-structured. The optional improvements above are just for robustness and edge cases.