**ArrayGraph Class Description**

The **ArrayGraph** class implements the **Graph** interface for an abstract data type (ADT) graph. It manages vertices and edges where each vertex is labelled, and edges connect pairs of vertices under certain constraints.

* **Instance Variables**: The class has two arrays: one for vertices (**Vertex<F>[]**) and one for edges (**Edge<F>[]**), both of generic type **F**. It also maintains two integer counters for the current numbers of vertices and edges, respectively.
* **Constructor**: Initializes the vertices and edges arrays with fixed sizes (20 for vertices, 50 for edges), using a generic array creation workaround due to Java's type erasure.
* **Graph Methods**:
  + **addEdge(Edge<F> e)**: Adds an edge if it doesn't already exist in the graph and if both vertices of the edge are present. It uses a linear search to check for the presence and uniqueness of the edge and vertices. After insertion, it sorts the edges array.
  + **addVertex(Vertex<F> v)**: Adds a vertex if it doesn't already exist in the graph, using a linear search for uniqueness checks. The vertices array is sorted after insertion.
  + **deleteEdge(Edge<F> e)**: Removes an edge if it exists, using a linear search for the edge and shifting elements to maintain array integrity.
  + **deleteVertex(Vertex<F> v)**: Removes a vertex and any edges connected to it. It performs a linear search for the vertex and edge deletion, with array element shifting as needed.
  + **vertexSet()**: Returns a set of all vertices.
  + **edgeSet()**: Returns a set of all edges.

**Time Complexity Analysis**

**addEdge(Edge<F> e)**

* **Underlying algorithms:** Linear search to check for the existence of the edge and vertices, and direct assignment for adding an edge.
* **Time complexity:** The method first checks if the edge already exists and if both vertices are present in the graph. This operation requires scanning through all edges and vertices, which takes *O*(*n*+*m*) time, where *n* is the number of vertices, and *m* is the number of edges. The sort operation at the end is not explicitly shown in the method but mentioned; assuming a simple comparison-based sort like bubble sort or insertion sort is used, it would have *O*(*m*2) complexity. However, the primary operation before sorting is linear.
* **Overall time complexity:** *O*(*n*+*m*+*m*2) due to the sort operation mentioned. If sorting is efficient or not performed every time, then *O*(*n*+*m*).

**addVertex(Vertex<F> v)**

* **Underlying algorithms:** Linear search for checking existence and direct assignment for adding a vertex.
* **Time complexity:** *O*(*n*) for scanning through the vertices to check for existence. Sorting the vertices is mentioned to maintain order, which could have a complexity of *O*(*n*2) for simple sorts.
* **Overall time complexity:** *O*(*n*2), primarily due to the sorting operation.

**deleteEdge(Edge<F> e)**

* **Underlying algorithms:** Linear search for edge existence and array manipulation for deletion.
* **Time complexity:** *O*(*m*) for finding the edge, and *O*(*m*) for shifting elements after deletion.
* **Overall time complexity:** *O*(*m*), as both operations are linear with respect to the number of edges.

**deleteVertex(Vertex<F> v)**

* **Underlying algorithms:** Linear search for vertex existence, array manipulation for vertex deletion, and nested linear search for edge deletion.
* **Time complexity:** *O*(*n*) for finding the vertex, *O*(*n*) for shifting vertices, and *O*(*m*) for each edge deletion check, leading to *O*(*mn*) if every edge is checked for each vertex deletion.
* **Overall time complexity:** *O*(*mn*+*n*), heavily influenced by the number of edges connected to the vertex being deleted.

**vertexSet() and edgeSet()**

* **Underlying algorithms:** Direct iteration over vertices and edges arrays.
* **Time complexity:** *O*(*n*) for **vertexSet()** and *O*(*m*) for **edgeSet()**, simply copying elements into a set.

**containsVertex(Vertex<F> v) and containsEdge(Edge<F> e)**

* **Underlying algorithms:** Linear search through vertices and edges.
* **Time complexity:** *O*(*n*) for **containsVertex** and *O*(*m*) for **containsEdge**.

**sortVertices() and sortEdges()**

* **Underlying algorithms:** Not explicitly detailed, but assuming a simple comparison-based sort like bubble sort or insertion sort.
* **Time complexity:** *O*(*n*2) for **sortVertices()** and *O*(*m*2) for **sortEdges()** if using simple sorting algorithms.

**Summary**

The **ArrayGraph** class relies on straightforward array manipulation techniques, which are simple but not always efficient. Adding, deleting, and searching operations have linear time complexities based on the size of the vertices or edges array they operate on. However, the sorting operations, which are critical for maintaining order in the graph, introduce quadratic time complexities *O*(*n*2) or *O*(*m*2)) that significantly impact the performance for large numbers of vertices or edges.

**Vertex and Edge Classes**

* **Vertex**: Encapsulates a value of generic type **F** and provides basic operations like getters, setters, and equality checks.
* **Edge**: Represents a directed edge between two vertices (start and end vertices), ensuring the start vertex label is smaller than the end vertex label.

**Recommendations for Description Document**

* **Class Descriptions**: Include brief descriptions of the **ArrayGraph**, **Vertex**, and **Edge** classes.
* **Method Signatures and Descriptions**: Detail the purpose and parameters of each method, along with its return type.
* **Complexity Analysis**: For each method, provide the time complexity analysis, explaining the rationale behind each calculation.
* **Use Cases**: Include examples of how each method is used, potentially using snippets from the **TestGraph** class.

**Conclusion**

The ArrayGraph project provides a foundational implementation of a graph ADT, using arrays to manage its vertices and edges. While this implementation serves the purpose of demonstrating the graph ADT's functionalities, it is not optimized for performance, particularly for large graphs, due to its reliance on arrays and the linear or quadratic time complexities of its operations. Future improvements could include the use of more efficient data structures, such as ArrayLists or HashMaps, to improve performance, especially for add and delete operations.