Sprint 3 Global Complexity Analysis

USEI17 - Build a PERT-CPM Graph: Complexity Analysis

ActivityReader Complexity Analysis

Method `readCSV()`

- **Complexity:** O(n + m)
- **Explanation:** The `readCSV` method processes a CSV file, parsing each line to extract activity data and build a graph. For each line, it reads and validates the information, adds activities as vertices, and establishes dependency relationships as edges. The complexity is O(n + m), where `n` is the number of activities (vertices) and `m` is the number of dependencies (edges). The reading and parsing operations are linear in the size of the file, while dependency processing scales with the number of relationships between activities.

Method `addFinishPredecessors()`

- **Complexity:** O(n)
- **Explanation:** This method iterates over all activities to add predecessors for the finish activity. It adds a predecessor for each activity to the finish task, requiring a single pass through the activities. As the number of activities is `n`, the time complexity of this method is O(n), where `n` represents the total number of activities.

Method `getFinalID()`

- **Complexity:** O(k)
- **Explanation:** This method extracts the numeric portion from an ID string. The time complexity is proportional to the length of the ID string, `k`, since it involves scanning and slicing the string. If the ID string is of length `k`, the operation takes O(k) time.

Method `validateString()`

- **Complexity:** O(1)
- **Explanation:** This method performs a simple validation to check if a string is neither null nor empty. Since the check is performed in constant time regardless of the string length, its complexity is O(1).

Method `validateParametersUnits()`

- **Complexity:** O(1)
- **Explanation:** This method performs basic validation for parameters like ID, duration, and cost, ensuring they meet expected formats and ranges. Each validation is a constant-time operation (checking types, non-null values, or ranges), resulting in O(1) complexity.

Method `checkString()`

- **Complexity:** O(1)
- **Explanation:** This method verifies that the given string is neither null nor empty. The check is a straightforward constant-time operation, so the complexity is O(1).

Method `checkConversion()`

- **Complexity:** O(1)
- **Explanation:** This method attempts to convert the provided string parameters into integers or doubles. Since the conversion process does not involve iteration over the string, it executes in constant time, resulting in O(1) complexity.

Method `checkConversionAndNegativeInt()`

- **Complexity:** O(1)
- **Explanation:** This method checks if a string can be successfully converted to a non-negative integer. The conversion and comparison are both constant-time operations, leading to O(1) complexity.

Method `checkConversionAndNegativeDouble()`

- **Complexity:** O(1)
- **Explanation:** Similar to the integer check, this method verifies if a string can be converted into a valid non-negative double. The operation remains constant time as it performs a single conversion and validation.

Method `sendNegativeError()`

- **Complexity:** O(1)
- **Explanation:** This method throws an exception if a parameter is found to be negative. Since throwing an exception is a constant-time operation, the complexity of this method is O(1).

Overall Complexity

- **Overall Complexity of `readCSV()`**: O(n + m)
- The most computationally expensive operation in the `ActivityReader` class is the reading and processing of the CSV file, which involves iterating through each activity (O(n)) and processing its dependencies (O(m)). Therefore, the overall complexity of this method is O(n + m), where `n` is the number of activities and `m` is the number of dependencies.

MapGraph Complexity Analysis

Method `MapGraph(boolean directed)`

- **Complexity:** O(1)
- **Explanation:** This constructor initializes a graph using a `LinkedHashMap` to store vertices and a boolean flag indicating whether the graph is directed. The initialization process only involves setting up the data structure and the flag, resulting in constant-time complexity, O(1).

Method `MapGraph(Graph<V, E> g)`

- **Complexity:** O(V + E)
- **Explanation:** This method copies an existing graph, iterating over all vertices (`V`) and edges (`E`) to recreate the graph's structure. Since the method needs to copy both vertices and edges, the complexity is O(V + E), where `V` is the number of vertices and `E` is the number of edges in the original graph.

Method `validVertex(V vert)`

- **Complexity:** O(1)
- **Explanation:** This method checks if a given vertex exists in the graph by performing a lookup in a `LinkedHashMap`. Since lookups in hash maps are generally constant time, the complexity is O(1).

Method `adjVertices(V vert)`

- **Complexity:** O(1)
- **Explanation:** This method retrieves the adjacent vertices of a given vertex. It looks up the adjacency list (or set) of the vertex, which is a constant-time operation, O(1).

Method `edges()`

- **Complexity:** O(V + E)
- **Explanation:** This method iterates over all vertices and collects their outgoing edges, resulting in a time complexity of O(V + E). The method processes all vertices (`V`) and traverses their associated edges (`E`), making the overall complexity proportional to the total number of vertices and edges in the graph.

Method `edge(V vOrig, V vDest)`

- **Complexity:** O(1)
- **Explanation:** This method retrieves the edge between two vertices. The lookup for the edge is typically done in constant time, assuming the graph is represented efficiently (e.g., with a hash map), leading to an O(1) complexity.

Method `outDegree(V vert)`

- **Complexity:** O(1)
- **Explanation:** This method returns the number of outgoing edges for a given vertex. Since the out-degree is typically stored or can be calculated in constant time, the complexity is O(1).

Method `inDegree(V vert)`

- **Complexity:** O(V)
- **Explanation:** This method calculates the in-degree by iterating over all vertices and counting how many have incoming edges to the specified vertex. This requires a full scan of the graph's vertices, making the complexity O(V), where `V` is the number of vertices in the graph.

Method `outgoingEdges(V vert)`

- **Complexity:** O(1)
- **Explanation:** This method retrieves the outgoing edges for a vertex, typically by performing a constant-time lookup in a hash map or adjacency list, resulting in O(1) complexity.

Method `incomingEdges(V vert)`

- **Complexity:** O(V)
- **Explanation:** Similar to `inDegree()`, this method calculates the incoming edges by scanning the graph's vertices to find all edges pointing to the specified vertex. This operation involves iterating through all vertices, resulting in a time complexity of O(V).

Method `addVertex(V vert)`

- **Complexity:** O(1)
- **Explanation:** This method adds a new vertex to the graph. Since adding a vertex to a graph generally involves inserting it into a data structure (e.g., a hash map

or list), the operation is typically constant-time, O(1).

Method `addEdge(V vOrig, V vDest, E weight)`

- **Complexity:** O(1) (amortized)
- **Explanation:** This method adds an edge between two vertices. The time complexity is generally considered O(1), assuming the graph structure supports efficient insertion. The operation involves adding an edge to the adjacency list or hash map, which typically occurs in constant time. If the graph's underlying data structure requires resizing or other overhead, the complexity may be amortized O(1).

Method `removeVertex(V vert)`

- **Complexity:** O(V + E)
- **Explanation:** This method removes a vertex from the graph, along with any edges connected to it. Removing a vertex involves iterating through all edges and adjusting the adjacency lists of other vertices, which results in O(V + E) complexity, where `V` is the number of vertices and `E` is the number of edges.

Method `removeEdge(V vOrig, V vDest)`

- **Complexity:** O(1) / O(1) for directed / undirected graphs
- **Explanation:** This method removes an edge between two vertices. The operation is typically constant time, O(1), for both directed and undirected graphs, assuming efficient access to the edge structure (e.g., a hash map or adjacency list).

Method `clone()`

- **Complexity:** O(V + E)
- **Explanation:** The `clone` method creates a deep copy of the graph, which involves copying both vertices and edges. Since each vertex and edge needs to be copied, the time complexity is O(V + E), where `V` is the number of vertices and `E` is the number of edges.

Method `toString(ID id)`

- **Complexity:** O(V + E)
- **Explanation:** This method generates a string representation of the graph by iterating over all vertices and edges. The time complexity is O(V + E), where `V` is the number of vertices and `E` is the number of edges.

Overall Complexity

- **Overall Complexity for MapGraph:** Most methods in the `MapGraph` class have **O(V + E)** complexity, where `V` is the number of vertices and `E` is the number of edges in the graph.

MapVertex Complexity Analysis

Method `MapVertex(V vert)`

- **Complexity:** O(1)
- **Explanation:** This constructor initializes the vertex and its adjacency map. The initialization process is done in constant time, O(1), as it simply sets up the data structure for storing adjacent vertices.

Method `getElement()`

- **Complexity:** O(1)
- **Explanation:** This method returns the stored vertex element. Since it directly accesses the element, the operation is constant-time, O(1).

Method `addAdjVert(V vAdj, Edge<V, E> edge)`

- **Complexity:** O(1)
- **Explanation:** This method adds an adjacent vertex and its associated edge to the adjacency map. Since inserting into a map is typically done in constant time, the complexity is O(1).

Method `remAdjVert(V vAdj)`

- **Complexity:** O(1)
- **Explanation:** This method removes an adjacent vertex from the adjacency map. The operation is done in constant time, O(1), since it directly removes the entry from the map.

Method `getEdge(V vAdj)`

- **Complexity:** O(1)
- **Explanation:** This method retrieves the edge associated with a given adjacent vertex. The lookup operation is constant time, O(1), as it accesses the edge directly from the map.

Method `numAdjVerts()`

- **Complexity:** O(1)
- **Explanation:** This method returns the number of adjacent vertices by simply returning the size of the adjacency map, a constant-time operation, O(1).

Method `getAllAdjVerts()`

- **Complexity:** O(k)
- **Explanation:** This method retrieves all adjacent vertices as a collection. The time complexity depends on the number of adjacent vertices (`k`), making it O(k), where `k` is the number of adjacent vertices for the specific vertex.

Method `getAllOutEdges()`

- **Complexity:** O(k)
- **Explanation:** This method retrieves all outgoing edges. The time complexity is proportional to the number of outgoing edges, which is O(k), where `k` is the number of outgoing edges for the specific vertex.

Method `toString()`

- **Complexity:** O(k)
- **Explanation:** This method generates a string representation of the vertex and its adjacent vertices. The complexity is O(k), where `k` is the number of adjacent vertices, as it iterates through them to build the string.

Algorithms Class Complexity Analysis

Method `hasCircularDependencies()`

- **Complexity:** O(V + E)
- **Explanation:** This method performs a Depth First Search (DFS) to detect cycles in the graph. The DFS algorithm visits each vertex (`V`) and edge (`E`) at most once, leading to a time complexity of O(V + E).

Method `hasCycleUtil()`

- **Complexity:** O(V + E)
- **Explanation:** This method is a utility function for DFS that helps check for cycles. Similar to `hasCircularDependencies()` , it visits vertices and edges once, resulting in a complexity of O(V + E).

Overall Complexity for Cycle Detection

- **Complexity:** O(V + E)
- **Explanation:** Both cycle detection methods in the `Algorithms` class involve depth-first traversal of the graph, and thus, their complexity is O(V + E), where `V` is the number of vertices and `E` is the number of edges.

Edge Class Complexity Analysis

Method `Edge()` (Constructor)

- **Complexity:** O(1)
- **Explanation:** This constructor initializes an edge with the origin vertex, destination vertex, and optional weight. The initialization of these attributes takes constant time, O(1).

Method `getVOrig()`

- **Complexity:** O(1)
- **Explanation:** This method returns the origin vertex of the edge. As it's a simple getter function, the operation is constant time, O(1).

Method `getVDest()`

- **Complexity:** O(1)
- **Explanation:** Similar to `getVOrig()` , this method returns the destination vertex of the edge, which takes constant time, O(1).

Method `getWeight()`

- **Complexity:** O(1)
- **Explanation:** This method returns the weight of the edge. The operation is a direct retrieval of the edge's weight attribute, so it runs in constant time, O(1).

Method `setWeight()`

- **Complexity:** O(1)

- **Explanation:** This method sets the weight of the edge. As setting a value in an attribute is a constant-time operation, the complexity is O(1).

Method `toString()`

- **Complexity:** O(1)
- **Explanation:** This method generates a string representation of the edge. It involves formatting and returning the string, which is done in constant time, O(1).

Method `equals()`

- **Complexity:** O(1)
- **Explanation:** This method compares two edges for equality. It checks if the origin, destination, and weight match, which are all constant-time comparisons, leading to O(1) complexity.

Summary of Overall Class Complexities

- **ActivityReader:** O(n + m)
- **MapGraph:** O(V + E)
- **MapVertex:** O(1) to O(k) (depending on the operation)
- **Algorithms:** O(V + E) for cycle detection methods
- **Edge:** O(1) for all methods

USEI18 - Detect Circular Dependencies: Complexity Analysis

Algorithms Class Complexity Analysis

Method	Complexity
`hasCircularDependencies()`	O(V + E)
`hasCycleUtil()`	O(V + E)

1. Method `hasCircularDependencies()`

- **Explanation:** The method `hasCircularDependencies` performs a Depth First Search (DFS) over the graph to detect cycles. It:
  - Iterates over all vertices in the graph, calling the `hasCycleUtil` helper method on each unvisited vertex.
  - The outer loop runs `V` times (where `V` is the number of vertices).
  - For each vertex, the helper function `hasCycleUtil` is invoked, which performs a DFS traversal, visiting vertices and their adjacent vertices.
  - The DFS will visit each vertex and edge at most once, so the time complexity for this traversal is **O(V + E)**, where:
    - `V` is the number of vertices (nodes).
    - `E` is the number of edges (connections between vertices).
- **Complexity: O(V + E)**
  - Since the method calls `hasCycleUtil` once for each unvisited vertex, the overall time complexity remains **O(V + E)**, where `V` is the number of vertices and `E` is the number of edges in the graph.

2. Method `hasCycleUtil()`

- **Explanation:** The helper function `hasCycleUtil` performs the actual DFS to detect cycles. It:
  - Marks the current vertex as visited and adds it to the recursion stack.
  - For each adjacent vertex:
    - If the adjacent vertex is already in the recursion stack, it indicates a cycle (since it means we're revisiting a vertex that has already been encountered during the current DFS path).
    - If the adjacent vertex is unvisited, the function recursively explores that vertex.
  - The DFS ensures that each vertex and edge is visited at most once during the traversal.
  - Since the function performs a DFS traversal and each vertex and edge is processed once, the complexity of this method is **O(V + E)** for each individual call.
- **Complexity: O(V + E)**
  - Each DFS call explores a connected component of the graph, so the time complexity of `hasCycleUtil` is proportional to the number of vertices and edges in that component. Since every call in the DFS results in visiting all vertices and edges once, the overall complexity for the graph is still **O(V + E)**.

Overall Complexity

- **Overall Complexity for Cycle Detection (Circular Dependencies): O(V + E)**
  - The cycle detection algorithm uses DFS to explore the graph and detect circular dependencies (cycles). The time complexity of DFS is proportional to the number of vertices (`V`) and edges (`E`), and since each vertex and edge is visited at most once, the total complexity for detecting cycles is **O(V + E)**.
  - This complexity is optimal for cycle detection in directed graphs, as each vertex and edge needs to be explored to determine if a cycle exists.

Summary

- **Time Complexity: O(V + E)**
  - **V:** The number of vertices (nodes) in the graph.
  - **E:** The number of edges (connections between vertices) in the graph.

USEI19 - Topological Sort of Project Activities: Complexity Analysis

Algorithms Class Complexity Analysis

Method	Complexity
`addEdge()`	O(1)
`write()`	O(N)
`performTopologicalSort()`	O(V+E)
`convertMapGraphToAdjacencyList()`	O(V+E)
`topologicalSort()`	O(V+E)

---

1. `addEdge()`

- **Explanation:**
  - The `addEdge` method adds a directed edge from one node to another in a graph and updates the in-degree of the target node.
  - The graph is represented using a map, where each node has a list of its neighbors.
  - **Steps in the Method:**
    - **`putIfAbsent` for initializing adjacency lists:** This operation initializes the adjacency list for the source (`from`) and target (`to`) nodes if they don't already exist. Since `putIfAbsent` operates in constant time, this step has a complexity of **O(1)**.
    - **Adding the edge to the adjacency list:** This operation simply appends the target node (`to`) to the adjacency list of the source node (`from`). The list insertion is **O(1)**.
    - **Updating the in-degree map:** The in-degree of the target node is incremented to indicate that the target has one more incoming edge. The map update is **O(1)**.
  - All these steps involve constant-time operations, and therefore the overall complexity for each call to `addEdge` is **O(1)**.
- **Complexity: O(1)**

---

2. `write()`

- **Explanation:**
  - The `write` method writes a list of strings (usually the project activities or topological order) to a file, one string per line.
  - **Steps in the Method:**
    - **Creating a `PrintWriter`:** This is a simple initialization of a writer object that allows writing to the specified file. Since creating the object doesn't depend on the file size or the number of strings, this step is **O(1)**.
    - **Writing the header line:** Writing a fixed header line to the file (e.g., "Activity ID, Start Time, End Time") is a constant-time operation, **O(1)**.
    - **Iterating over the list and writing each string:** The method iterates over the `verticesList` , writing each string (activity information). Since the list contains `N` strings, where `N` is the number of strings, the total complexity for this loop is **O(N)**.
    - **Closing the `PrintWriter`:** This operation is **O(1)** as it just closes the file stream.
  - **Time Complexity:** The most time-consuming part of this method is writing all the strings to the file, resulting in a complexity of **O(N)**.
  - **Space Complexity:** Since no new large data structures are created, the space complexity remains **O(1)**.
- **Complexity: O(N)**, where `N` is the number of strings in the list.

---

3. `performTopologicalSort()`

- **Explanation:**
  - This method orchestrates the topological sort of the graph by first converting it into an adjacency list and then performing the sort.
  - **Steps in the Method:**
    - **Convert graph to adjacency list (O(V + E)):** The graph representation (which could be a map, adjacency matrix, etc.) is converted into an adjacency list format. Each vertex is processed once (O(V)), and each edge is processed once (O(E)), leading to a complexity of **O(V + E)** for this step.
    - **Adding edges for each connection (O(E)):** Once the graph is in adjacency list format, the method will iterate over all edges and add them to the

appropriate lists.
        - **Performing the topological sort (O(V + E)):** The actual sorting is done using Kahn's algorithm, which processes each vertex and each edge, leading to a time complexity of **O(V + E)**.
    - **Time Complexity:** The overall complexity for `performTopologicalSort()` is dominated by the graph traversal and sorting, resulting in **O(V + E)**.
    - **Space Complexity:** The space complexity here is mainly for storing the adjacency list and in-degree map. Both of these require space proportional to the number of vertices (`V`) and edges (`E`), leading to **O(V + E)** space complexity.

- **Complexity: O(V + E)**, where `V` is the number of vertices and `E` is the number of edges.

---

4. `convertMapGraphToAdjacencyList()`

- **Explanation:**

    - This method converts a graph, represented as a MapGraph, into an adjacency list, which is often a more efficient structure for graph algorithms like topological sorting.
    - **Steps in the Method:**
        - **Iterating over all vertices (O(V)):** The method first iterates over all vertices in the graph to ensure that each vertex has an adjacency list.
        - **Processing neighbors (O(E)):** For each vertex, it retrieves the neighbors (edges) and adds them to the adjacency list. Since the graph is sparse, this step takes **O(E)**.
    - **Time Complexity:** The method iterates over all vertices once and over all edges once, resulting in a total complexity of **O(V + E)**.
    - **Space Complexity:** The space required for the adjacency list is proportional to the number of vertices and edges, so the space complexity is also **O(V + E)**.

- **Complexity: O(V + E)**, where `V` is the number of vertices and `E` is the number of edges.

---

5. `topologicalSort()`

- **Explanation:**

    - The `topologicalSort()` method performs the actual topological sorting using Kahn’s Algorithm, which uses in-degree counting and queue-based processing.
    - **Steps in the Method:**
        - **Initializing a queue (O(V)):** The method iterates over all vertices to find those with zero in-degree, which are then added to a queue. This step is **O(V)**.
        - **Processing each node in the queue (O(E)):** The method processes each node in the queue, iterating over its neighbors and decrementing their in-degrees. The total time to process all neighbors is proportional to the number of edges, **O(E)**.
        - **Cycle detection (O(1)):** To detect cycles, the method compares the size of the sorted list to the number of vertices. If they don't match, a cycle is detected. This comparison is a constant-time operation, **O(1)**.
    - **Time Complexity:** The overall time complexity for Kahn's Algorithm is **O(V + E)**, as each vertex and edge is processed once.
    - **Space Complexity:** The space complexity is dominated by the space needed for the queue, which holds at most **V ** nodes, and the adjacency list and in-degree map, leading to **O(V + E)** space complexity.

- **Complexity: O(V + E)**, where `V` is the number of vertices and `E` is the number of edges.

Overall Complexity

- The overall complexity of the algorithm depends on the dominant operations involved in processing the graph. The graph traversal and edge handling (i.e., adjacency list construction and topological sorting) are the key factors, and these involve processing all vertices and edges.
- File writing (if included in the flow) is dependent on the number of strings being written, and it does not affect the core graph traversal complexity.
- Therefore, the overall complexity for the entire system is **O(V + E)** for graph processing, with file writing contributing an additional **O(N)** if needed.

USEI20 - Project Scheduling

ProjectSchedule Class Complexity Analysis

---

1. `generateVerticesListBFS`

- **Objective**: This method generates a topologically ordered list of vertices (activities).
- **Operations**:
    1. **Clear the list**: `verticesList.clear()` . This operation is `O(n)` , where `n` is the number of activities, because it removes all elements from the list.
    2. **Generate topological order**: `Algorithms.getTopologicalOrder(projectGraph)` . The complexity of this operation depends on the implementation of the topological sorting algorithm. If it uses depth-first search (DFS) or breadth-first search (BFS), the time complexity will be `O(V + E)` , where `V` is the number of vertices (activities) and `E` is the number of edges (dependencies between activities).

Complexity:

- **Time Complexity**: `O(V + E)`
    - The complexity is dominated by the graph traversal (topological sort).

- **Space Complexity**: `O(V)`
  - The space complexity is required to store the list of vertices in topological order.

---

2. `calculateScheduleAnalysis`

- **Objective**: This method calculates scheduling metrics such as `earliestStart`, `earliestFinish`, `latestStart`, `latestFinish`, and `slack` for each activity.
- **Operations**:
  1. **Generate topologically ordered list of vertices**: This has already been analyzed in the complexity of `generateVerticesListBFS`, being `O(V + E)`.
  2. **Calculate `earliestStart` and `earliestFinish` for each activity**:
     - Iterating over all vertices is `O(V)`.
     - For each activity, the method checks its predecessors to calculate the `earliestStart`. This involves iterating through the list of predecessors for each activity, leading to a time complexity of `O(p)` for each activity, where `p` is the number of predecessors. In the worst case, `p = V`, so this step has a complexity of `O(V)` for each activity.
     - Calculating `earliestFinish` is a constant-time operation for each activity, i.e., `O(1)`.
     - Overall, calculating the `earliestStart` and `earliestFinish` for all activities has a complexity of `O(V * p)` in the worst case.
  3. **Calculate `latestFinish`, `latestStart`, and `slack` for each activity**:
     - This step iterates through the list of activities in reverse order (topologically sorted list). This iteration takes `O(V)`.
     - For each activity, the method checks its successors to calculate the `latestStart`. This involves iterating through the list of successors, which in the worst case could be `O(V)` for each activity.
     - Calculating `slack` is a constant-time operation for each activity, i.e., `O(1)`.
     - Overall, calculating `latestFinish`, `latestStart`, and `slack` has a complexity of `O(V * s)` in the worst case, where `s` is the number of successors.

Complexity:

- **Time Complexity**: `O(V + E)`
  - The complexity is dominated by the graph traversal (topological sort) and the analysis for each activity.
- **Space Complexity**: `O(V)`
  - The space complexity is used to store the scheduling information (earliest and latest times, slack, etc.) for each activity.

---

3. `getMaxEarliestFinish`

- **Objective**: Finds the maximum `earliestFinish` time among the predecessors of a given activity.
- **Operations**:
  1. **Iterating through predecessors**: This involves iterating through the list of predecessors of the activity. In the worst case, there are `V` predecessors, so this operation has a time complexity of `O(V)`.
  2. **Finding the maximum `earliestFinish`**: This is done in constant time for each predecessor, i.e., `O(1)`.

Complexity:

- **Time Complexity**: `O(p)` where `p` is the number of predecessors (in the worst case, `p = V`).
- **Space Complexity**: `O(1)`
  - No additional space is required beyond the current activity's predecessors.

---

4. `getMinLatestStart`

- **Objective**: Finds the minimum `latestStart` time among the successors of a given activity.
- **Operations**:
  1. **Iterating through successors**: This involves iterating through the list of successors of the activity. In the worst case, there are `V` successors, so this operation has a time complexity of `O(V)`.
  2. **Finding the minimum `latestStart`**: This is done in constant time for each successor, i.e., `O(1)`.

Complexity:

- **Time Complexity**: `O(s)` where `s` is the number of successors (in the worst case, `s = V`).
- **Space Complexity**: `O(1)`
  - No additional space is required beyond the current activity's successors.

Final Complexity Table

Method	Time Complexity	Space Complexity
`generateVerticesListBFS`	`O(V + E)`	`O(V)`
`calculateScheduleAnalysis`	`O(V + E)`	`O(V)`

Method	Time Complexity	Space Complexity
`getMaxEarliestFinish`	`O(p)` (where p = V)	`O(1)`
`getMinLatestStart`	`O(s)` (where s = V)	`O(1)`

Conclusions

- **Time Complexity**: The overall time complexity for the most significant methods is dominated by the graph traversal and dependency analysis. The time complexity for these operations is **O(V + E)**, typical for graph-related algorithms that process each vertex and edge.
- **Space Complexity**: The space complexity for each method is generally **O(V)**, which is necessary to store the list of activities and their associated scheduling data, such as `earliestStart`, `earliestFinish`, and slack for each activity.
- **Efficiency**: The auxiliary methods like `getMaxEarliestFinish` and `getMinLatestStart` have linear complexity with respect to the number of predecessors or successors but do not add significant complexity beyond the main scheduling analysis.

The `ProjectSchedule` class provides an efficient implementation with linear time and space complexities relative to the number of activities and dependencies, ensuring that the scheduling and project analysis processes scale well with larger projects.

USEI21 - Export Project Schedule: Complexity Analysis

ProjectSchedule Class Complexity Analysis

---

1. `sendProjectScheduleToFile`

The `sendProjectScheduleToFile` method is responsible for exporting the project schedule to a CSV file, including activity details and their dependencies. Let's break down its complexity step by step.

Operations:

1. **Calculate Schedule Analysis**:
   - The method calls `calculateScheduleAnalysis()`, which has been previously analyzed with a complexity of `O(V + E)`.
2. **File Writing with PrintWriter**:
   - **File Creation**: The creation of a `PrintWriter` to write data to a CSV file is a constant-time operation: `O(1)`.
   - **Writing Header**: Writing the header line is a constant-time operation: `O(1)`.
3. **Writing Activity Data**:
   - The loop iterates over `verticesList` (activities), writing data for each activity. The size of `verticesList` is `V` (the number of activities), so the loop will iterate `V` times.
   - For each activity, several operations are performed:
     - Writing activity details (ID, cost, duration, timings) is a constant-time operation for each activity: `O(1)`.
     - Iterating through the activity's predecessors (`getPredecessors`) is done for each activity. In the worst case, each activity can have up to `V` predecessors, leading to a time complexity of `O(V)` for the predecessor iteration.
   - Therefore, the time complexity for writing activity data for all activities becomes `O(V * p)`, where `p` is the number of predecessors per activity. In the worst case, this becomes `O(V^2)`.
4. **Writing Special Activities (Start and Finish)**:
   - Writing the start and finish activities is a constant-time operation (`O(1)`), as they are handled separately from the main loop.
5. **Closing the PrintWriter**:
   - Closing the `PrintWriter` is a constant-time operation: `O(1)`.

Complexity Analysis:

Time Complexity:

- **File writing operations**: The loop over `verticesList` involves iterating over `V` activities and writing their details. The worst-case time complexity for writing each activity involves iterating over all its predecessors, so the time complexity is:
  - `O(V + V * p)` or `O(V^2)` in the worst case.
  - The calculation of schedule analysis is `O(V + E)`, but this has already been handled separately.

Thus, the overall time complexity of the method is:

- **Time Complexity**: `O(V^2)` (due to iterating over predecessors for each activity).

Space Complexity:

- The space complexity is determined by the storage of the activity data and the temporary storage used by the `PrintWriter`.
- Since the method only writes to a file and does not store additional large data structures beyond the existing project schedule, the space complexity is:
    - **Space Complexity**: `O(1)` (for file writing, as no additional large data structures are created during the export).

Final Complexity Table

Method	Time Complexity	Space Complexity
`sendProjectScheduleToFile`	`O(V^2)`	`O(1)`

Conclusions

- **Efficiency**: The method is efficient in terms of space, but its time complexity can become quadratic (`O(V^2)`) in the worst case due to the iteration over each activity's predecessors. However, for typical project schedules with fewer dependencies, this should perform well.
- **Scalability**: As the number of activities (`V`) increases, the method's time complexity grows quadratically, which may impact performance for very large projects with numerous dependencies. Optimizing the handling of predecessors could reduce this complexity in future improvements.
- **Error Handling**: The method includes basic error handling for file writing issues, which improves its robustness. However, further improvements in error reporting and logging could enhance its reliability in larger-scale or production environments.

In conclusion, while the `sendProjectScheduleToFile` method provides valuable functionality for exporting project schedules, attention should be paid to its scalability with large numbers of activities and dependencies.

USEI21 - Export Project Schedule

Complexity Analysis of the `sendProjectScheduleToFile` Method

The `sendProjectScheduleToFile` method exports the project schedule to a CSV file, including activity details and their dependencies. Below is a detailed analysis of its time and space complexity.

Step-by-Step Complexity Breakdown

1. Schedule Analysis Calculation

- **Operation**: Calls `calculateScheduleAnalysis()`.
- **Complexity**: This method has been analyzed to have a time complexity of **`O(V + E)`**, where `V` is the number of activities (vertices) and `E` is the number of dependencies (edges).
- **Result**: **`O(V + E)`**.

---

2. File Writing Operations

File Creation

- **Operation**: Instantiates a `PrintWriter` for the output file.
- **Complexity**: Constant time operation.
- **Result**: **`O(1)`**.

Writing the Header

- **Operation**: Writes the CSV header line.
- **Complexity**: Constant time operation.
- **Result**: **`O(1)`**.

---

3. Writing Activity Data

- **Operation**: Loops over `verticesList` (list of activities). For each activity, it writes details and iterates through its predecessors.

Outer Loop

- **Description**: Iterates over all activities.

- **Complexity**: `O(V)`, where `V` is the number of activities.

Inner Loop (Predecessors)

- **Description**: For each activity, iterates through its predecessors to write dependencies.
- **Worst Case**: Each activity has up to `V` predecessors, resulting in `O(V)` complexity for the inner loop.
- **Combined Complexity for All Activities**:
  Outer loop (`O(V)`) × Inner loop (`O(V)` for predecessors) = `O(V^2)`.

---

4. Writing Special Activities (Start and Finish)

- **Operation**: Separately handles "Start" and "Finish" nodes.
- **Complexity**: Constant time operation.
- **Result**: `O(1)`.

---

5. Closing the File

- **Operation**: Closes the `PrintWriter`.
- **Complexity**: Constant time operation.
- **Result**: `O(1)`.

Final Complexity Analysis

Step	Time Complexity	Space Complexity
Schedule Analysis Calculation	`O(V + E)`	`O(1)`
File Creation and Header Write	`O(1)`	`O(1)`
Writing Activity Data	`O(V^2)`	`O(1)`
Writing Special Activities	`O(1)`	`O(1)`
File Closing	`O(1)`	`O(1)`
Total	**`O(V^2)`**	**`O(1)`**

Conclusions

Efficiency

- **Time Complexity**:
  - The method's dominant factor is the nested loop over activities and their predecessors, resulting in `O(V^2)` in the worst case.
  - The schedule analysis step adds `O(V + E)`, but it does not dominate the overall complexity.
- **Space Complexity**: The method is efficient in terms of space, as it writes directly to a file and does not require additional large data structures. The overall space complexity is `O(1)`.

Scalability

- The method may experience performance degradation for large projects with a high number of activities and dense dependencies, as the `O(V^2)` complexity can become significant. However, for projects with sparse dependencies, the performance impact is less pronounced.

Error Handling

- The method includes basic error handling for file writing operations, ensuring robustness in most scenarios. Improvements could be made by enhancing logging or exception propagation for better debugging in production environments.

Optimization Opportunities

1. **Reduce Predecessor Iteration Cost**:
   - Using adjacency lists or hash maps for faster lookup of predecessors could reduce the inner loop's complexity.

2. **Parallelization**:
    - Writing activity data could be parallelized if activities are independent of one another.
3. **Batched File Writing**:
    - Writing data in larger batches instead of individual lines could improve I/O performance for very large datasets.

Summary

The `sendProjectScheduleToFile` method is efficient in terms of space but has a worst-case time complexity of ** `O(V^2)` ** due to nested loops over activities and their dependencies. While suitable for moderately sized projects, it may require optimization to handle large and densely connected schedules effectively.

USEI22 - Critical Path Calculation: Complexity Analysis

Method Overview

The `calculateCriticalPath` method computes the critical path and total project duration in a graph of activities. It leverages depth-first search (DFS) traversal, using stacks for iterative exploration of paths. The method identifies nodes connected to a designated "Start" node and evaluates paths ending at nodes connected to a "Finish" node.

Line-by-Line Complexity Analysis

1. Input Validation

- **Description**: The method checks whether the input graph is null or empty.
- **Complexity**:
    - Null check is **O(1)**.
    - Checking if vertices are empty involves traversing all vertices, resulting in **O(V)**.
- **Result**: **O(V)**.

---

2. Initialization of Result Variables

- **Description**: Allocates space for the critical path list and initializes the total duration variable.
- **Complexity**: Initialization operations are constant time.
- **Result**: **O(1)**.

---

3. Filter Valid Activities

- **Description**: Iterates over all vertices in the graph, excluding the "Start" and "Finish" nodes from the filtered activity list.
- **Complexity**:
    - Iterates through all vertices: **O(V)**.
    - Performs constant-time checks for each vertex.
- **Result**: **O(V)**.

---

4. Identify Start Nodes

- **Description**: Iterates through the filtered activities to identify nodes connected to the "Start" node.
    - For each node, the method checks its predecessors for the presence of a "Start" node.
- **Complexity**:
    - Outer loop iterates through filtered activities: **O(V)**.
    - Inner loop iterates through predecessors: **O(E)**.
- **Result**: **O(V * E)**.

---

5. Initialize Stacks for DFS

- **Description**: Initializes three stacks used for iterative DFS traversal—one for nodes, one for path durations, and one for tracking paths.
- **Complexity**: Stack initialization is a constant-time operation.
- **Result**: **O(1)**.

---

6. Explore Paths Using DFS

Outer Loop for Start Nodes

- **Description**: Iterates over all start nodes to initialize DFS traversal for each.
- **Complexity**: For all start nodes, the outer loop is **O(V)**.

DFS Iteration

- **Description**: Iteratively pops nodes from the stack, evaluates them, and explores successors.
    - Each node is processed once, and its stack operations are constant time: **O(1)**.
- **Complexity**:
    - Processing all nodes via DFS traversal: **O(V)**.

End Node Check

- **Description**: For each node, checks if it connects to a "Finish" node by examining its successors.
- **Complexity**:
    - Checking successors for each node: **O(E)** in total.
- **Result**: **O(E)**.

Exploring Valid Successors

- **Description**: Iterates through filtered activities to identify valid successors. Successors must have zero slack and be listed as a successor of the current node.
- **Complexity**:
    - Outer loop iterates through filtered activities: **O(V)**.
    - For each activity, checks if it's a successor: **O(1)**.
- **Result**: **O(V^2)** in the worst case.

---

7. Construct Results

- **Description**: Stores the calculated critical path and total duration in a result map.
- **Complexity**: Inserting results into a map is a constant-time operation.
- **Result**: **O(1)**.

---

8. Error Handling

- **Description**: Catches exceptions during critical path calculation and logs an error message.
- **Complexity**: Exception handling is a constant-time operation.
- **Result**: **O(1)**.

Final Complexity Summary

Step	Time Complexity
Input Validation	O(V)
Initialization of Variables	O(1)
Filter Valid Activities	O(V)
Identify Start Nodes	O(V * E)
Initialize Stacks	O(1)
DFS Path Exploration	O(V^2)
Construct Results	O(1)
Error Handling	O(1)
Total Complexity	**O(V^2)**

Conclusion

Efficiency

- The method relies on iterative DFS traversal to compute paths. The main computational bottleneck lies in nested loops over vertices and their predecessors/successors, leading to **O(V^2)** complexity in the worst case.

Scalability

- **Suitable for Moderate Graphs**: The method performs well for small to medium-sized graphs but may struggle with large datasets due to quadratic complexity.
- **Graph Size**: The number of vertices (`V`) and edges (`E`) heavily influences performance.

Optimization Opportunities

1. **Adjacency Lists or Hash Maps**:
   - Using adjacency lists or hash maps for faster lookups could reduce nested loop dependencies.
2. **Parallel DFS**:
   - Parallelizing DFS for independent start nodes may improve performance on large graphs.
3. **Path Caching**:
   - Reuse precomputed paths or intermediate results where applicable to avoid redundant calculations.

Error Handling

- The current error-handling mechanism is basic, relying on logging. Introducing structured logging or exception propagation could improve maintainability and debugging.

USEI24 - Simulate Project Delays and Their Impact: Complexity Analysis

Algorithms Class Complexity Analysis

Method	Time Complexity
`calculateCriticalPath()`	O(1)
`updateActivityDuration()`	O(1)
`removeActivities()`	O(V + E)

---

1. `calculateCriticalPath()`

Description:

This method is responsible for determining the critical path in a project schedule. Instead of performing the calculation directly, it delegates the task to `CriticalPath.calculateCriticalPath(createdMap)`, which processes the graph to find the critical path.

Operations:

1. **Delegation**:
   - Calls the method `CriticalPath.calculateCriticalPath`, passing a graph object (`createdMap`) as input. This operation itself is constant time: **O(1)**.
   - The actual complexity of the operation depends on the implementation of the `CriticalPath.calculateCriticalPath` method, which is typically **O(V + E)**, where `V` is the number of vertices (activities) and `E` is the number of edges (dependencies).

Complexity:

- **Time Complexity**: O(1) for this method itself. The actual critical path calculation (`CriticalPath.calculateCriticalPath`) has a separate complexity of **O(V + E)**.

---

2. `updateActivityDuration()`

Description:

This method updates the duration of an activity by adding a specified value. If the new duration becomes negative, it ensures the duration is reset to zero.

Operations:

1. **Retrieve Current Duration**:
    - Retrieves the activity's current duration with a getter method: **O(1)**.
2. **Update Duration**:
    - Adds the specified value to the current duration: **O(1)**.
    - Checks if the new duration is non-negative: **O(1)**.
3. **Set New Duration**:
    - Updates the duration using a setter method: **O(1)**.

Complexity:

- **Time Complexity**: **O(1)**. All operations are constant time and do not depend on the size of the input.
- **Space Complexity**: **O(1)**. No additional data structures are created.

---

3. `removeActivities()`

Description:

This method removes specified vertices (activities) from the graph. It ensures that dependencies connected to the removed activities are also handled appropriately.

Operations:

1. **Identify Activities to Remove**:
    - Iterates through all vertices (`V`) in the graph to identify which ones need to be removed. This involves retrieving each vertex's ID and comparing it with the IDs of the activities to remove: **O(V)**.
2. **Remove Vertices**:
    - Iterates through the list of identified activities to remove. For each vertex:
        - Removes it from the graph structure.
        - Updates adjacency lists or internal data structures to maintain the graph's integrity. This step requires traversing the edges connected to each removed vertex: **O(E)**.

Complexity:

- **Time Complexity**: **O(V + E)**. Iterating over all vertices and edges in the graph dominates the method's performance.
- **Space Complexity**: **O(1)**. The method does not use additional space beyond existing structures.

Overall Complexity Analysis

Time Complexity

1. **Graph Traversal**:

    - The most computationally expensive operations involve graph traversal. Removing activities (`removeActivities`) and calculating the critical path (`calculateCriticalPath`) involve visiting all vertices and edges.
    - The dominant time complexity for the system is **O(V + E)**, where `V` is the number of vertices and `E` is the number of edges.

2. **Activity Updates**:

    - Updating activity durations is a constant-time operation (**O(1)**) and does not significantly contribute to the overall complexity.

Space Complexity

- The methods primarily operate directly on the existing graph data structure and do not create large intermediate data structures.
- The space complexity for all methods is **O(1)**.

Conclusion

- The overall system is dominated by the graph operations (`calculateCriticalPath` and `removeActivities`), which have a time complexity of **O(V + E)**.
- The `updateActivityDuration` method is efficient and has a negligible impact on the system's overall performance.
- The implementation is space-efficient, as no additional memory is required for intermediate calculations.

While the system handles small- to medium-sized projects efficiently, larger graphs with high numbers of vertices and edges could result in performance bottlenecks. Future improvements might involve optimizing graph traversal or parallelizing operations for better scalability.