GIVEN:
An arbitrary workflow, i.e. directed acyclic graph (DAG) G=(V, E), where each node v is a job and each directed edge (u, v) shows that the output data of job u is transferred as the input data of job v, K homogeneous machines, the execution time t(v) of each job running individually on a single machine, and the communication time t(u, v) of each data transfer between jobs u and v running in different machines.

QUESTION:
Find the minimum execution time of the entire workflow along with a feasible schedule mapping all the jobs onto no more than K machines

REQUIREMENT:
Comment on the difficulty (i.e., computational complexity) of the above problem, design an efficient algorithm, implement the solution, (which may be an exact, approximation, or heuristic algorithm), and show the execution results. (C/C++ in Linux and Makefile are preferred; C/C++/Java is required.)

I do not expect a perfect solution from you. Based on your source code and algorithm description, I will evaluate
1) whether you have a correct, clear idea (algorithm) to solve this optimization problem.
2) whether your implementation is correct in source files.
3) your code readability (such as code format and necessary comments), algorithm efficiency, and self-learning capacity.

REPORT:

Algorithm Overview
- Represents workflow as a directed weighted graph
- Vertices are jobs, edges are dependencies
- Edge weights represent data transfer times

Steps to solve the algorithm:

1. **Find the order of job sequencing:**
   The algorithm employs topological sorting to determine the order of job execution. It ensures that a job is scheduled only after its dependencies have been completed. The topological sorting is performed using a modified Breadth-First Search (BFS) approach.

2. **Schedule the Jobs:**
   The algorithm schedules jobs onto machines based on their dependencies and execution times. It calculates the earliest available machine for a job, considering the completion times of its dependencies. The scheduling is done in a way that minimizes the makespan, i.e., the total execution time of the entire workflow.

3. **Dependency Consideration:**
   The algorithm appropriately considers dependencies when scheduling jobs. It calculates the finish time of each job based on the completion times of its dependent jobs and communication times.

Analysis time complexity:
1. Topological sorting of the jobs:

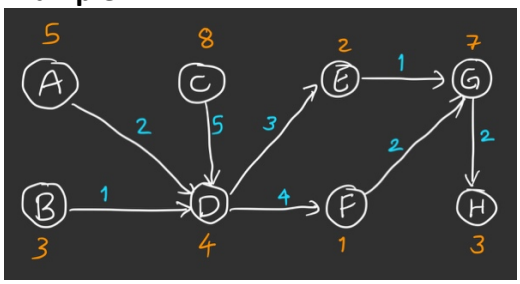
- This can be done in O(V+E) time, where V is the number of jobs (vertices) and E is the number of dependencies (edges).

2. Scheduling each job:
   - Find predecessors: O(V*E)
   - Finding the earliest machine takes O(K) time, where K is the number of machines.
   - Find and update finish time based on dependencies take maximum O(V)
   - Overall time complexity for scheduling each job is O(V*E + K + V) = O(V*E + K)

3. The overall algorithm does the following:
   - Topological sort: O(V+E)
   - Schedule V jobs: O(V(V*E+K))

So the overall time complexity is O(V+E + V(V*E+K)) = O(VVE + K)

Example:



Consider the above workflow graph.
A, B, and C are not dependent on any jobs. But D can only start the job when A, B, and C are completed. Similarly, E is dependent on D, F is dependent on D, G is dependent on E and F, and H is dependent on G.

Working of Algorithm:

1. Initialize an array of length equal to the number of machines (k) to store finishing times for each machine.

2. Visit each vertex in the topological order.

3. For each vertex, identify its predecessors (vertices that provide input to the current job). Since we use a topological order, the predecessors' jobs are guaranteed to be completed before the current job.

   For example, if we consider D, the predecessors of D are A, B, and C.

4. Determine the maximum time for computation and data transfer among all predecessor jobs.

   'A' is completed at 5 and can transfer data from A -> D in 2, so in total 'A' takes 7 units of time.
   'B' is completed at 3 and can transfer data from B -> D in 1, so in total 'B' takes 4 units of time.
   'C' is completed at 8 and can transfer data from C -> D in 5, so in total 'C' takes 13 units of time.

5. Add the determined maximum value to the current job's completion time, compare it with the finishing time of the earliest machine, and store the maximum of these two values.

D can complete 4 units of time, ‘D’ can only start at 13, so ‘D’ will complete its job at 17.

6. Repeat this process for all vertices.

7. Among all the machine completion times, select the maximum time. This represents the minimum time required to complete the workflow with K machines.

Programming Language: Java
NOTE: I also tried to do it in CPP, but since I am better at Java I codded in Java. But if you require me to know CPP, I assure you that I will learn CPP before the start of next semester. Your guidance is appreciated.
Thank You

Code:

```
import java.util.*;

// Workflow scheduling class
class WorkflowScheduling {

    // Weighted graph to represent workflow
    static class WeightedGraph {
        // Map of vertices by unique id
        Map<Character, Vertex> vertices = new HashMap<>();

        // Add vertex with id and duration
        void addVertex(char id, int duration) {
            vertices.put(id, new Vertex(id, duration));
        }

        // Add weighted edge between vertices
        void addEdge(char u, char v, int weight) {
            vertices.get(u).outputs.add(new Edge(vertices.get(v), weight));
        }

        // Get underlying vertices map
        Map<Character, Vertex> getVertices() {
            return vertices;
        }

        // Vertex representing a job
        static class Vertex {
            // Job id - Vertex name
            char id;
            // Job duration for a particular vertex
            int duration;
            // To know whether the vertex is visited ot not
            boolean visited = false;

            // Completion time after scheduling
            int completionTime = -1;
            // Outgoing edges to dependents
            List<Edge> outputs = new ArrayList<>();

            // Constructor
            Vertex(char id, int duration) {
                this.id = id;
                this.duration = duration;
            }
```

```
        // Get the time of a vertex when it completes its job.
        public int getCompletionTime(){
            return completionTime;
        }


        // String representation of an instant
        @Override
        public String toString() {
            return String.valueOf(id);
        }
    }

    // Directed edge representing dependencies
    static class Edge {
        // Destination vertex
        Vertex dest;

        // Dependency weight as data transfer time
        int weight;

        // Constructor
        Edge(Vertex dest, int weight) {
            this.dest = dest;
            this.weight = weight;
        }
        @Override
        public String toString() {
            return "(" + dest.id + ", " + weight + ")";
        }
    }
}

// Topologically sort graph vertices
public static List<Character> topologicalSort(WeightedGraph graph) {
    List<Character> result = new ArrayList<>();
    Queue<WeightedGraph.Vertex> queue = new LinkedList<>();

    // Calculate in-degrees for each vertex
    int[] inDegrees = new int[graph.getVertices().size()];
    for (WeightedGraph.Vertex v : graph.getVertices().values()) {
        for (WeightedGraph.Edge e : v.outputs) {
            inDegrees[e.dest.id - 'A']++;
        }
    }

    // Enqueue vertices with in-degree 0
    for (WeightedGraph.Vertex v : graph.getVertices().values()) {
        if (inDegrees[v.id - 'A'] == 0) {
            queue.offer(v);
        }
    }

    // BFS traversal to produce topological order
    while (!queue.isEmpty()) {
        WeightedGraph.Vertex v = queue.poll();
        result.add(v.id);

        for (WeightedGraph.Edge e : v.outputs) {
            // Decrease in-degree of the destination vertex
            inDegrees[e.dest.id - 'A']--;
```

```
                // If in-degree becomes 0, enqueue the destination vertex
                if (inDegrees[e.dest.id - 'A'] == 0) {
                    queue.offer(e.dest);
                }
            }
        }

        return result;
    }

    // Schedule jobs on machines
    public static int schedule(WeightedGraph graph, int numMachines) {

        // No machine for jobs scheduling
        if(numMachines <= 0)
            return 0;

        // Schedule jobs on a single machine
        if (numMachines == 1) {
            int totalDuration = 0;

            // Calculate sum of all vertices' durations
            for (WorkflowScheduling.WeightedGraph.Vertex vertex :
graph.getVertices().values()) {
                totalDuration += vertex.duration;
            }

            // Calculate sum of all edges' weights
            int totalEdgeWeights = 0;
            for (WorkflowScheduling.WeightedGraph.Vertex vertex :
graph.getVertices().values()) {
                for (WorkflowScheduling.WeightedGraph.Edge edge : vertex.outputs) {
                    totalEdgeWeights += edge.weight;
                }
            }

            int totalWeight = totalDuration + totalEdgeWeights;

            // Return the total weight
            return totalWeight;
        }

        // Get topological order of jobs
        List<Character> order = topologicalSort(graph);

        // Print the Topological order
        System.out.println("\nTopological sort (Job execution order): "+order);
        System.out.println();

        // Track scheduled vertices for each machine
        List<List<Character>> scheduledVertices = new ArrayList<>();
        for (int i = 0; i < numMachines; i++) {
            scheduledVertices.add(new ArrayList<>());
        }

        // Initialize machine finish time
        int[] machineFinishTime = new int[numMachines];

        // Schedule jobs one by one
        for (char jobId : order) {

            // Get job vertex
            WeightedGraph.Vertex job = graph.getVertices().get(jobId);
```

```
            // Find the predecessors of job vertex
            List<WeightedGraph.Vertex> inputs = getPredecessors(graph, job);
            System.out.println("\nPredecessors of "+jobId+" are: "+inputs);
            System.out.println();

            // Find the machine with earliest finish time
            int earliestMachine = findEarliestMachine(machineFinishTime);
            int maxDependencyFinishTime = 0;

            // Update finish time based on dependencies
            for (WeightedGraph.Vertex in : inputs) {

                WeightedGraph.Edge edge = getEdge(graph, in, job);

                int dependencyFinishTime =
Math.max(in.getCompletionTime(),machineFinishTime[earliestMachine])+edge.weight;
                System.out.println("Dependency "+ in +" Finish Time: "+
dependencyFinishTime);
                maxDependencyFinishTime = Math.max(maxDependencyFinishTime,
dependencyFinishTime);
                System.out.println("Maximum Dependency Finish Time:
"+maxDependencyFinishTime);
            }

            // No predecessors
            if(inputs.size()==0){

                // Update finish time with job duration
                machineFinishTime[earliestMachine] +=  job.duration;

                // Set job completion time
                job.completionTime = machineFinishTime[earliestMachine];
            }
            else {

                // Update finish time with job duration
                machineFinishTime[earliestMachine] =
Math.max(maxDependencyFinishTime,machineFinishTime[earliestMachine]) + job.duration;

                // Set job completion time
                job.completionTime = maxDependencyFinishTime + job.duration;

            }

            // Track scheduled vertices for the machine
            scheduledVertices.get(earliestMachine).add(job.id);
//            System.out.println(scheduledVertices);
            System.out.print("\nMachines Finish time: [ ");
            for(int m=0;m<numMachines;m++){
                System.out.print(machineFinishTime[m]+" ");
            }
            System.out.println("]");
            System.out.println("------");
        }

        // Print vertices scheduled on each machine
        for (int i = 0; i < numMachines; i++) {
            System.out.println("Machine " + (i + 1) + " scheduled vertices: " +
scheduledVertices.get(i));
        }
```

```
        // Get overall makespan
        int makespan = Arrays.stream(machineFinishTime).max().orElse(0);
        return makespan;
    }

    // Find the machine with the earliest finish time
    private static int findEarliestMachine(int[] machineFinishTime) {
        int earliestMachine = 0;
        for (int i = 1; i < machineFinishTime.length; i++) {
            if (machineFinishTime[i] < machineFinishTime[earliestMachine]) {
                earliestMachine = i;
            }
        }
        return earliestMachine;
    }

    // Get all the input vertices of a particular vertex
    public static List<WeightedGraph.Vertex> getPredecessors(WeightedGraph graph,
WeightedGraph.Vertex vertex) {
        List<WeightedGraph.Vertex> predecessors = new ArrayList<>();
        for (WeightedGraph.Vertex v : graph.getVertices().values()) {
            for (WeightedGraph.Edge edge : v.outputs) {
                if (edge.dest == vertex) {
                    predecessors.add(v);
                }
            }
        }
        return predecessors;
    }

    // Get edge between two vertices
    public static WeightedGraph.Edge getEdge(WeightedGraph graph, WeightedGraph.Vertex
source, WeightedGraph.Vertex destination) {
        for (WeightedGraph.Edge edge : source.outputs) {
            if (edge.dest == destination) {
                return edge;
            }
        }
        return null;
    }


    // Main driver method
    public static void main(String[] args) {

        // Create workflow graph
        WeightedGraph graph = new WeightedGraph();

        // Adding vertices and edges - Add jobs and dependencies

        // Example-1
        graph.addVertex('A', 5);
        graph.addVertex('B', 3);
        graph.addVertex('C', 8);
        graph.addVertex('D', 4);
        graph.addVertex('E', 2);
        graph.addVertex('F', 1);
        graph.addVertex('G', 7);
        graph.addVertex('H', 3);
        graph.addEdge('A', 'D', 2);
        graph.addEdge('B', 'D', 1);
        graph.addEdge('C', 'D', 5);
        graph.addEdge('D', 'E', 3);
```

```
        graph.addEdge('D', 'F', 4);
        graph.addEdge('E', 'G', 1);
        graph.addEdge('F', 'G', 2);
        graph.addEdge('G', 'H', 2);

        // Example -2
//        graph.addVertex('A', 2);
//        graph.addVertex('B', 3);
//        graph.addVertex('C', 4);
//        graph.addVertex('D', 9);
//        graph.addVertex('E', 7);
//        graph.addVertex('F', 3);
//        graph.addVertex('G', 2);
//        graph.addVertex('H', 3);
//        graph.addVertex('I', 5);
//        graph.addVertex('J', 7);
//        graph.addEdge('A', 'D', 3);
//        graph.addEdge('B', 'D', 2);
//        graph.addEdge('B', 'E', 3);
//        graph.addEdge('C', 'E', 2);
//        graph.addEdge('D', 'F', 1);
//        graph.addEdge('E', 'G', 5);
//        graph.addEdge('E', 'H', 2);
//        graph.addEdge('F', 'I', 3);
//        graph.addEdge('G', 'J', 3);
//        graph.addEdge('H', 'I', 3);
//        graph.addEdge('H', 'J', 3);

        //Example-3
//        graph.addVertex('A', 3);
//        graph.addVertex('B', 3);
//        graph.addVertex('C', 3);
//        graph.addVertex('D', 3);
//        graph.addVertex('E', 3);
//        graph.addVertex('F', 3);
//        graph.addVertex('G', 3);
//        graph.addVertex('H', 3);
//        graph.addVertex('I', 3);
//        graph.addVertex('J', 3);
//        graph.addVertex('K', 3);
//        graph.addVertex('L', 3);
//        graph.addVertex('M', 3);
//        graph.addEdge('C', 'A', 2);
//        graph.addEdge('C', 'B', 2);
//        graph.addEdge('D', 'B', 2);
//        graph.addEdge('D', 'G', 2);
//        graph.addEdge('D', 'H', 2);
//        graph.addEdge('E', 'A', 2);
//        graph.addEdge('E', 'D', 2);
//        graph.addEdge('E', 'F', 2);
//        graph.addEdge('F', 'K', 2);
//        graph.addEdge('F', 'J', 2);
//        graph.addEdge('G', 'I', 2);
//        graph.addEdge('H', 'I', 2);
//        graph.addEdge('J', 'I', 2);
//        graph.addEdge('J', 'L', 2);
//        graph.addEdge('J', 'M', 2);
//        graph.addEdge('K', 'J', 2);

        // Set the number of machines
        int numMachines = 3;
        int makespan = schedule(graph, numMachines);
```

```
        if(numMachines>1) {
            // Print vertices with completion times
            System.out.println("\nJob Completion Times:");
            for (WeightedGraph.Vertex vertex : graph.getVertices().values()) {
                System.out.println(vertex.id + ": " + vertex.completionTime);
            }
        }

        // Print results
        System.out.println("\nMinimum execution time of entire workflow: " + makespan);

    }
}
```

Output:

Topological sort (Job execution order): [A, B, C, D, E, F, G, H]

Predecessors of A are: []

Machines Finish time: [5 0 0]

Predecessors of B are: []

Machines Finish time: [5 3 0]

Predecessors of C are: []

Machines Finish time: [5 3 8]

Predecessors of D are: [A, B, C]

Dependency A Finish Time: 7
Maximum Dependency Finish Time: 7
Dependency B Finish Time: 4
Maximum Dependency Finish Time: 7
Dependency C Finish Time: 13
Maximum Dependency Finish Time: 13

Machines Finish time: [5 17 8]

Predecessors of E are: [D]

Dependency D Finish Time: 20
Maximum Dependency Finish Time: 20

Machines Finish time: [22 17 8]

Predecessors of F are: [D]

Dependency D Finish Time: 21
Maximum Dependency Finish Time: 21

Machines Finish time: [22 17 22]

Predecessors of G are: [E, F]

Dependency E Finish Time: 23
Maximum Dependency Finish Time: 23
Dependency F Finish Time: 24
Maximum Dependency Finish Time: 24

Machines Finish time: [22 31 22]

Predecessors of H are: [G]

Dependency G Finish Time: 33
Maximum Dependency Finish Time: 33

Machines Finish time: [36 31 22]

Machine 1 scheduled vertices: [A, E, H]
Machine 2 scheduled vertices: [B, D, G]
Machine 3 scheduled vertices: [C, F]

Job Completion Times:
A: 5
B: 3
C: 8
D: 17
E: 22
F: 22
G: 31
H: 36

Minimum execution time of entire workflow: 36