WIA1002 DATA STRUCTURE
LAB TEST 3
Duration: 1 hour
(Friday, 7th June 2024, 7.00am – 8.00am)

Introduction

Modern scientific collaborations have opened up the opportunity of solving complex problems that involve multidisciplinary expertise and large-scale computational experiments. These experiments comprise a sequence of processing steps, or referred as tasks, that need to be executed on selected computing platforms. Each task takes input data, either from preceding tasks or data sources, performs predefined computations, and produces output data for the succeeding tasks or to be delivered to data storage. Tasks are usually separate instances of executable programs that need to be run in a predefined order. A common strategy to make the experiments more manageable is to model the processing steps as workflows, as shown in the Figure 1, and use a workflow management system to organise the enactment.

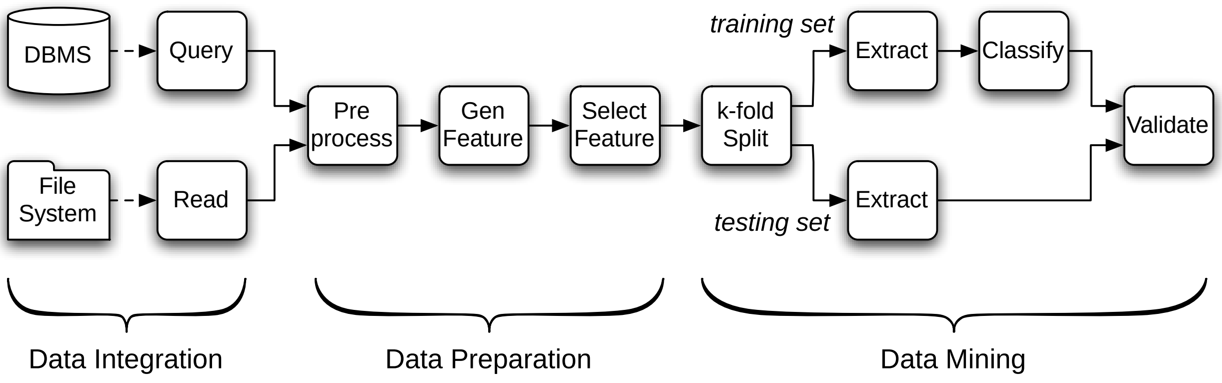

Figure 1

A workflow comprises three components: a list of tasks/operations, the dependencies between the interconnected tasks (the flow), and the data resources. In a graph representation, the tasks and data resources are the vertices, and the dependencies are the edges connecting the vertices. The tasks are the operations and the edges indicating the operations' order. The example workflow demonstrates the basic workflow patterns. The three tasks in the data-preparation stage form a sequence, which is executed one after another following the direction of the arrow on the edges. The sequence is then split into two parallel threads in the data mining stage. This gives the potential to run both threads in parallel. These threads are then merged back into a single sequence towards the end of the process.

Managing the execution of the workflow is a challenge. Over the past decades, many workflow management systems had been developed to support the scientific communities in managing their complex and large-scale experiments. One of the popular platform is HTCondor[1]. HTCondor is a system for dynamically sharing computational resources between competing

[1] HTCondor: https://htcondor.org/

computational tasks. As an HTCondor user, you will describe your computational tasks as a series of independent, asynchronous “jobs.” Each job will be described in a job submission file, that specifies the executable (i.e. the program), the input and output files, together with other specifications needed for the execution. The file ends with a keyword queue, which trigger the command to submit this job into the platform for execution.

Sample job submission .sub file:

```
# sleep.sub -- simple sleep job
executable = sleep.sh
log = sleep.log
output = sleep.out
error = sleep.err

should_transfer_files = Yes
when_to_transfer_output = ON_EXIT

request_cpus = 1
request_memory = 512M
request_disk = 1G
estimate_runtime = 12

queue
```

The workflow is represented as a Directed Acyclic Graph (DAG)[2] as shown in Figure 2. HTCondor has a component called DAGMan[3] that handles the DAG execution in the platform.

$$\begin{tikzcd}
 & \boxed{A (10)} \arrow[dl] \arrow[dr] & \\
\boxed{B (5)} \arrow[dr] & & \boxed{C (20)} \arrow[dl] \\
 & \boxed{D (15)} &
\end{tikzcd}$$

Figure 2

A DAGMan workflow is described in a **DAG input file**. The input file specifies the nodes of the DAG as well as the dependencies that order the DAG, as shown below:

```
JOB A A.sub
JOB B B.sub
JOB C C.sub
JOB D D.sub
```

[2] Directed Acyclic Graph (DAG): https://en.wikipedia.org/wiki/Directed_acyclic_graph
[3] DAGMan: https://htcondor.readthedocs.io/en/latest/automated-workflows/dagman-introduction.html

```
PARENT A CHILD B C
PARENT B C CHILD D
```

The `PARENT A CHILD B C` indicate that task A needs to be executed before task B and task C.

Your task:

1. Graph Data Structure Implementation:
   a. Implement a graph where each node represents a task.
   b. Each task has a task name, an estimated run time, and dependencies.
2. File I/O:
   a. Read the input DAG file and .sub files, and store the data in the graph data structure.
3. Graph Traversal and Sorting:
   a. Traverse the graph to read task information.
   b. Print out the tasks sorted by their run time in ascending order.
4. Find the Critical Path:
   a. Implement an algorithm to find the critical path, which is the path with the longest execution time from the start to the end of the workflow.

Input File Format

1. The input DAG file will be a plain text file where each line represents a task and its dependencies. The format of the file is as follows:
   a. JOB task_name task_file: Specifies a task and its corresponding .sub file.
   b. PARENT task_name CHILD task_name: Specifies dependencies between tasks.
2. Each job submission .sub file contains several lines of configuration, including a line specifying the estimated run time for the task in the format estimate_runtime = <time_in_minutes>.

Requirements

1. **Graph Node Class**:
   - Create a class Task with the following properties:
     - String taskName
     - int estimatedRunTime
     - List<String> dependencies
2. **Graph Class**:
   - Create a class WorkflowGraph that:

        - Stores the tasks and their dependencies using lists.
        - Reads the input DAG file and .sub files, and constructs the graph.
        - Provides methods for graph traversal and finding the critical path.
        - Provides a method to print tasks sorted by their estimated run time.

3. **Main Program**:
    - The main program should:
        - Read the input DAG file specified as a command-line argument.
        - Construct the WorkflowGraph.
        - Print the tasks in ascending order of their run times.
        - Print the critical path and its total run time.

Example Output

Given the example input files, the expected output would be:

```
Tasks sorted by estimated run time:
Task B : 5
Task A : 10
Task D : 15
Task C : 20


Critical Path: Task A -> Task C -> Task D
Total Run Time: 45
```