EECS2011A-F20: Fundamentals of Data Structures

Mini Project (25% + 5% Bonus)

Due: December 13, 2020 @ 11:59pm

Quick Q&A (read carefully)

Who?

Solo project; just you.

Why?

I want you to master the data structures and algorithms discussed in the class by leveraging them to solve real-world problems.

What?

We will focus on a common scheduling problem in cloud computing. You will have to check the validity of the application workflow, find a proper order of execution for functions, and try to reduce the total execution time of the application as much as possible.

Constraint:

- Development in Java (compatible with JDK 14) **without** using any built-in/third-party data structures or algorithms.
- Your implementation for each part should use no more that 256MB of memory.
- The execution time for each part should be less than 10 seconds.

Grading criteria

The grades are based on four criteria.

- Correctness: give the working solution within the memory and time limit (256MB, 10s)
- Result: try to make the total execution time of the application workflow as short as possible
- Performance: try to have the best time complexity for your implementation within the memory limitation (i.e. no more than 256MB)
- Implementation Report: explain and analyze your implementation in the PDF report

More questions?

Please submit your questions on the course forum.

Introduction

Workflow

A large cloud application is often composed of many small parts. Each small part is a **function** (method), which is a block of code that performs a specific task. For each function, it takes a certain amount of time to execute. We refer to the execution duration of a function as its **response time**. Of course, the computer needs to execute all functions to get the output of the application.

The order of execution for functions **cannot** be random since there are dependencies among functions. For example in Figure 1, the function f_2 takes the output of the function f_1 as its input. In this case, f_2 can be triggered only after the execution of f_1 is completed. We can use a graph, called **workflow**, to abstract such **dependencies** and the orchestration of functions in the application.

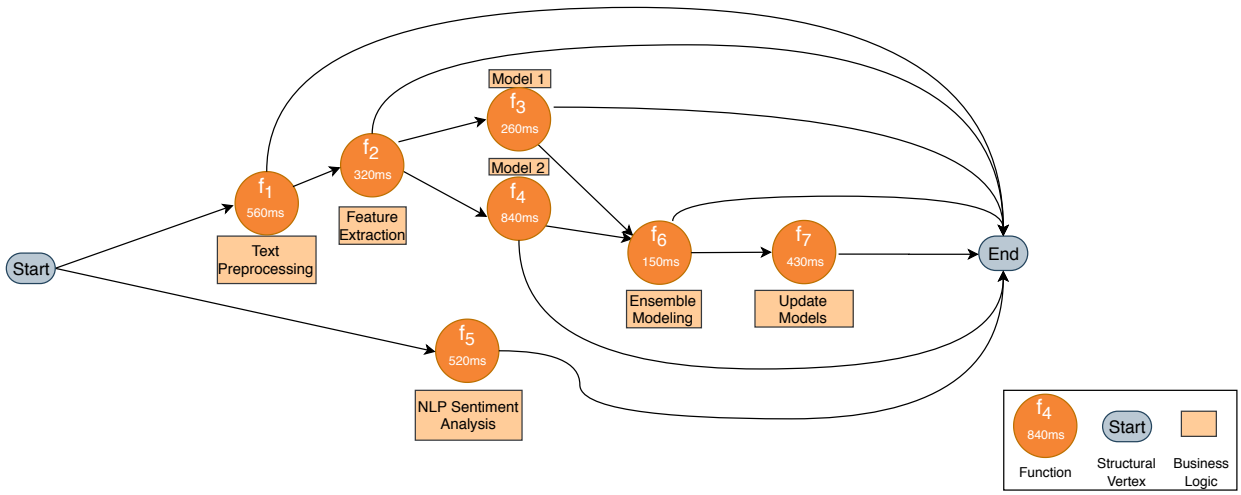

Figure 1: The workflow of an application for natural language processing and text mining

Figure 1 gives an example of a workflow. In the workflow graph, **functions** and **dependencies** are abstracted as **vertices** and **directed edges**, respectively. For instance, $\boxed{f_1} \rightarrow \boxed{f_2}$ represents the function f_2 takes the output of the function f_1 as its input. The number in milliseconds in each function is its response time.

There are two structural vertices in the workflow that are not functions: "Start" and "End." The "Start" vertex just represents the entry point of the application. If a function has only one dependency, which is the entry point, it basically means the function does not require any outputs from other functions. The "End" vertex represents the endpoint of the application. All functions are the dependencies of the endpoint, because the application is completed only when all functions are executed. Specifically, we refer to the amount of time taken up executing all functions as **the total execution time of the application**.

Distributed Computing

For applications composed of compute-intensive functions, we assume only **one function** can be executed at any point in time on **one machine**, since each compute-intensive function gets nearly full CPU time. We can use multiple machines to speed up the application execution as multiple functions can be executed simultaneously on them.

Cloud computing platforms, such as AWS, Azure, and GCP, provide cloud instances. A cloud instance can be viewed as a standalone machine with its own CPU. We can easily launch a number of identical instances on the cloud, connect them through proper communication protocols, and create a cluster. In other words, a cluster is a system composed of a number of independent machines, and we can refer to it as

a distributed system. By allocating functions to different instances, multiple functions can be executed at the same time in a distributed way, thus speeding up the application execution.

In the following questions, we may use such distributed systems composed of multiple **identical** machines to host applications. For brevity, we do **not** consider any delay incurred by distributed computing. Note that a cluster with X machines can execute **at most** X functions simultaneously.

Part 1: Validity of the workflow [7 points]

In this part you need to check if the workflow is valid. A valid workflow graph should satisfy the following conditions

- The workflow should be a directed acyclic graph (DAG) without loops and cycles.
- The "End" vertex is reachable from every function, and every function is reachable from the "Start" vertex (no disconnected vertices).

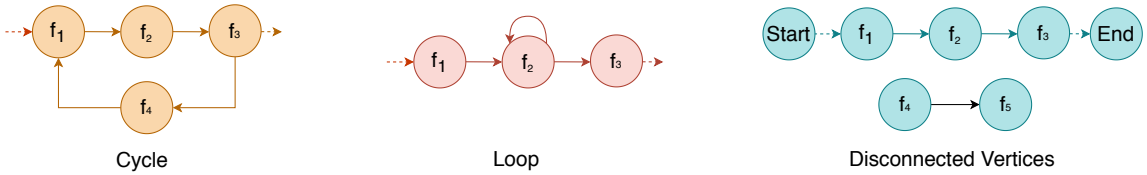

Figure 2: Sample **illegal** structures

Figure 2 depicts sample illegal structures. You need to develop a method ***check_validity()*** to check the validity of the given workflow. The method should print True if the workflow is valid, otherwise False.

Sample input

The input has **(2n+3)** lines, where n is the number of functions in the workflow. The first line contains an integer n ($1 \leq n \leq 100$), the number of functions in the workflow. The next **(n+2)** lines contain a $(n+2) \times (n+2)$ matrix describing the orchestration of the workflow. The rows of the matrix correspond to $[Start, f_1, f_2, ..., f_{n-1}, f_n, End]$, and so do the columns. If there is an edge going from f_u to f_v, the number on the $(u+1)$-th row and $(v+1)$-th column in the matrix should be 1. The next **n** lines contain a series of numbers describing the response time in ms for each function in the workflow. The number on the i-th line in the series is the response time of f_i.

```
4
0 1 1 1 0 0
0 0 0 0 1 1
0 0 0 0 0 1
0 0 1 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0
23
92
168
94
```

Sample output

Print out **True** if the workflow is **valid**, and print **False** if the input is **not** a valid workflow.

```
True
```

Part 2: Schedule functions on a single machine [8 points]

Let us assume we only have one machine. In other words, only one function can be executed at the time. Here, you need to find a proper order of execution for functions in the application and try to make the total execution time as short as possible. You need to implement a method named ***schedule_1()*** to give your solution.

Sample input

The input format is same as Part 1.

```
4
0 1 0 0 0 0
0 0 1 1 1 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 1 0 0 1
0 0 0 0 0 0
124
149
66
171
```

Sample output

The output is composed of **(n+1)** lines, where n is the number of functions in the workflow. The first **n** lines should each contain two space separated integers representing the name of the function and the time to start its execution, respectively. The last line shows an integer representing the total execution time of the application. Note that there might be multiple valid execution orders for functions.

```
1 0
4 124
2 295
3 444
510
```

Part 3: Schedule functions on a cluster with the same size of the application [10 points]

For an application with n functions, we create a cluster with n machines. You need to find a proper order of execution for functions on this cluster and try to make the total execution time as short as possible. For this part you need to implement a method named ***schedule_x()*** to give your solution.

Sample input

The input format is same as Part 1.

```
4
0 1 0 0 0 0
0 0 1 1 1 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 1 0 0 1
0 0 0 0 0 0
124
149
66
171
```

Sample output

The output format is same as Part 2.

```
1 0
2 295
3 124
4 124
444
```

Part 4: Optional [5 points]

This part is optional and you won't lose any point if you don't do it. However, you will receive 5 extra credits (i.e., 5% of the total course weight) as bonus if you implement it correctly.

Schedule functions on a cluster with 2 machines

In this part, you need to find a proper order of execution for functions on a cluster with 2 machines and try to make the total execution time as short as possible. For this part you will implement a method named ***schedule_2()*** to give your solution.

Sample input

The input format is same as Part 1.

```
4
0 1 0 0 0 0
0 0 1 1 1 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 1 0 0 1
0 0 0 0 0 0
124
149
66
171
```

Sample output

The output format is same as Part 2.

```
1 0
2 295
3 124
4 124
444
```

Implementation Instructions:

Here are some guidelines and tips for your implementations:

1. You need to start with the GitHub repository **starter kit** that has been created for this project.
2. In the starter kit, you will see a package named `eecs2011.project` in which, you will find two Java files. The `Solution.java` class is where you implement your solutions.
3. You may **NOT** change the method declarations in the `Solution` class nor the package name. You can add any number of classes to the package to implement your solution.
4. You should read data from the standard input and write your answer to the standard output. There is an example in the `Sample` class showing **one possible way** to read input, store data, and write answers using standard input/output. You are encouraged to implement your input/output/storage methods for better performance.
5. Please do **NOT** read and write data (i.e., interim results) from and to the disk. Such operations are not allowed and will be blocked.
6. You should strictly follow the sample output format. Implementations that do not follow the correct format will be marked as 0 by the judge system.
7. We use similar methods as most online judge systems to check your solutions. We use pipelines to send input to your program and send your output to our programming judge system. You can check if your code can get correct input and produce the right output using the pipes command:
   `cat sample_input | java yourclass.java` (Linux or macOS)
   `type sample_input | java yourclass.java` (Windows)

Deliverables:

One zip file with the name of `mini_project.zip` including 2 files:

1. Your Java source codes (src folder) zipped in one file: `source_code_your-student-id.zip` For example, if your student ID is 123456, you should name the source code zipped file as `source_code_123456.zip`
2. A short PDF final report including your implementation details: `final_report.pdf`

Your final report should include the following for each part of the project:

- pseudocode of your solution
- list the name of algorithms and data structures used in your solution
- a short explanation of why you use such algorithms and data structures
- time complexities of your solution in big-O notation