MIE562F 2020 Coding Assignment #1

Total Marks: 10

All code must be written by you alone. It is an academic offense to submit code written by someone else and to write code that someone else submits.

Late submissions are not accepted.

The assignment is to write Python 3 code to do the following:

- read in a job shop scheduling problem file (with a specified file format – see below)
- solve the problem using the shortest processing time (SPT) dispatch rule
- return the makespan and solution found

We have done our best to answer all questions in this document. Please read it carefully. If you have a question that is not covered here (and you are sure), please post it to piazza.

1 Problem Definition Details

You can find definitions of the job shop scheduling problem and the SPT dispatch rule in the course slides and/or textbook. In this section, we provide details on how we represent the problem instances and the conventions used.

1.1 Instance Files

Each instance file consists of the following:

- a line of description
- a line containing the number of jobs and the number of machines
- then one line for each job, listing the machine number and processing time for each step of the job.

The machines are implicitly numbered starting at 0.
Here is an example instance file.

```
Sample Instance
2 3
2  1  0  3  1  6
0  8  1  5  2 10
```

In this instance there are 2 jobs and 3 machines. Job 0 is on line 3 and job 1 line 4. Job 0, operation 0 requires machine 2 and has a processing time of 1. Job 0, operation 1 requires machine 0 and has a processing time of 3. And so on.

This format follows the standard JSP instance format (see `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt`) and so you should be able to find many test problems for your code.

A small set of instances, with answers, has been posted for you on Quercus (see Coding Assignment #1). You are strongly encouraged to create (or find) your own instances to test your code.

1.2 Job Numbering

Jobs are numbered starting at zero. Each job row in the instance corresponds to a job with the numbering starting at zero and incrementing by 1 with each line (e.g., the row after Job 0 contains Job 1). Operations within each job are numbered starting at zero; again they increment by one. As such, an operation can be uniquely identified by the pair of its job number and operation number within its job. So the first operation in Job 0 is (0,0), the second operation is (0,1), and so on.

1.3 Tie Breaking

It is important that you follow this tie breaking rule to ensure that your algorithm finds the same solutions as the one we have implemented and will compare against.

In your algorithm, you must use the SPT dispatch rule to order the list of available operations on each machine. Depending on the data, you may have to order two or more operations with the same processing time. If this happens, you are guaranteed that each operation will come from a different job.[1] **Therefore, to break ties, you should order the operations in ascending order of job number.**

For example, if you have operation (3,2) (i.e., job 3, operation 2) and operation (4,0) both with processing time 10 and both available to be executed on the same machine, then (3,2) should come before (4,0) in your list because $3 < 4$.

2 Coding and Marking Details

We will use the automated marking system called MarkUS to mark your code (see "Marking Scheme" below). MarkUS is running Python 3.8.3 and your code needs to work with this version of Python.[2]

In this section, we provide the coding requirements and other necessary information for using the MarkUS grading system.

2.1 Function Definition

You must submit a Python file called `coding1.py` that contains (perhaps among other functions) a function with the following signature:

> `run_SPT(instance)`, where `instance` is a string of the filename of the instance (e.g., "ft06.txt").

[1] I will leave it to you to figure out why this is true.

[2] Generally, as long as you are using Python 3.5 or greater, you should not have a problem.

The function should read in the problem instance (assume it is in the same folder as the `coding1.py` file) and solve the problem with the SPT dispatch rule with tie breaking as described above.

The function should return a tuple containing:

- the makespan: integer
- a schedule: a dictionary with keys corresponding to the job number and values being a list of start times for that job's operations following that job's operation order.

For example the solution to the "Sample Instance" given above is:

```
(23, {0: [0, 8, 13], 1: [0, 8, 13]})
```

The makespan is 23. Job 0, operation 0 starts at time 0. Job 0, operation 1 starts at time 8. Etc.

MarkUS will call your function and parse the return value to compare it against the correct answer. It is important that you follow the above instructions exactly.

Do not put any `input` or `print` statements in your code.

2.2 Use of External Modules: Don't

You may not use any external modules for your code. The reasons for this is that they really will not help you, we want you to write the algorithm not depend on some other implementation, and MarkUS will crash if you try to use any modules that we have not installed.

If you have a good argument as to why you should be allowed to use a particular external module that does not defeat the purpose of this assignment, I am willing to listen to it. Please email Prof. Beck.

2.3 Marking Scheme

The marks for this project will be distributed as follows:

- 7 marks from MarkUS. MarkUS will call your `run_SPT` function with seven different problem instances. If your code returns the correct answer for an instance, you will get 1 mark. If it doesn't return the correct answer, you will get 0 marks for that instance.
- 3 marks from code review. Your code will be manually reviewed (i.e., by a human) and awarded a mark out of 3 based on the following scale:

  0. no code submitted or the code submitted doesn't even try to solve the problem;
  1. code submitted but horrible;
  2. code is pretty good but a few ugly/painful parts;
  3. good code.

Once marks are released, you will have one week to submit a remark request to Prof Beck. Such a request must specify the reason you believe that your code has been marked incorrectly. (E.g., "The correct answer to test #3 is X. The output of my code in MarkUS is X but I received 0 marks for that instance.").

2.4 Submitting Your Code

By the assignment due date, you must submit your python code file (`coding1.py`) at: `https://markus.engineering.utoronto.ca/mie562` by logging to MarkUS with your UTORID and password.

To submit your solution file, follow the instructions below.

- On your MarkUS homepage, click the assignment name to go to its submission page.
- Click the "Submissions" tab at the top of the page.
- Click "Add A New File". A new row will be added to the table. Click "Choose File" or "Submit" in the new row. (The button's name will depend on your web browser.) In the window that appears, browse to where you have saved your file and select it. Click the button.
- Your submitted file will now appear in the table. Check that the filename is correct (i.e., `coding1.py`). If you click on the filename, you will see the contents of the file. You should check that the contents are correct, and that you have submitted what you wanted to submit.

MAKE SURE THE FILENAME IS CORRECT (`coding1.py`).
MARKUS IS CASE SENSITIVE

If you are unable to submit your code to MarkUS, email Victor (vic.lo@mail.utoronto.ca) for assistance. If there are technical problems with MarkUS at the deadline (e.g., MarkUS is down) you should email your code to Victor and Prof. Beck. If it arrives before the due date and the systems issues are confirmed, your submission will be considered on-time.

2.5 MarkUS Sanity Check

As much as possible, we want to remove errors due to technical issues with MarkUS and your use of it. Therefore, we have provided 5 test instances *that will not be graded* but that you can use to ensure that your code will run within the MarkUS system.

To run non-graded tests:

- Log into MarkUS (see above)
- Click the 'Automated Testing' tab
- Click 'Run Tests' and wait 10-20 seconds then refresh the page
- Check your auto-marked grade out of 5. This is not your assignment grade, this is only for you. You can submit your code an unlimited times before the deadline, but you can only use the auto-tester once per hour.

You should create/find your own test cases and try them on your own computer. The unreleased test cases will contain corner cases that are not included in the released test cases. You should not rely on the MarkUS as a testing framework. Again, auto-test is available for ONE TEST PER HOUR.

3 Academic Integrity

This is an individual assignment and you are, therefore, expected to complete it entirely by yourself.

We will be using code checking software called MOSS (`https://theory.stanford.edu/~aiken/moss/`) to compare the code you submit with all the other submissions. MOSS is a sophisticated algorithm for detecting similarities in the structure of your code. When MOSS flags significant similarity among submitted code, we manually compare the code and if, in our judgment, there is evidence of plagiarism, we will proceed with the University mandated academic offense procedures.

We will also be monitoring sites such as Chegg and others for the appearance of this assignment. If you are a Chegg user, you should be aware that your user agreement allows Chegg to share your personal information with the University of Toronto for suspected infringements of academic integrity. In the last academic year (in the first year Python programming course, which I also teach), a number of penalties for academic offences were assessed via such information.

Please see the course syllabus for more information on academic integrity. If you have questions, please contact Prof. Beck.

I apologize if the above stresses you out. The reality is that most of you don't cheat and, therefore, for most of you, this section is completely irrelevant. If you have no intention of cheating, you have nothing to worry about.