FJSSP

Flexible Job Shop Scheduling Problem

Practical Work II

Subject

Advanced Data Structures

Teachers

Luis Ferreira , João Silva

Student

João Rodrigues, 16928

LESI

Summary

Present report is meant to document an academic work.

The project consists of a final part of solving scheduling problems of FJSSP(*Flexible Job Shop Scheduling Problem)*. On the first part, it was developed a starting approach.

The implementation is made in C programming language.

Index

Contextualization

As mentioned on summary, this project has academic purposes, having origin on a practical work to be evaluated on the 1st year subject, **Advanced Data Structures**, from the course LESI – Bachelor on Computer Systems Engineering.

Project consists of applying content taught on the subject's class, where the student should be able to adapt the advanced data structures.

Problem in hands

On first part it was implemented basis for a resolution of scheduling an FJSSP(*Flexible Job Shop Scheduling Problem*), where it wasn't thinked how to schedule, but to store basic data. At this point, the idea is to actually schedule all jobs, give an output and if possible, make a representation of the data.

Requirements

The enunciation of the project asks for the follow:

1. Definition of a dynamic data structure to represent a finite set of jobs, where to each job, is associated a set of operations
2. Storing and Reading of text files representing a *process plan*.
3. Insertion of a new job
4. Removal of a job
5. Insertion of a new operation on a job
6. Removal of a certain operation of a job
7. Edit an operation from a job
8. Proposal of a resolution to the scheduling problem
9. Representation of different *process plans*, having respective scheduling proposals.

Scheduling Research

Tree for each Job

At first, it was meant to use trees to store each job's possibilities to be completed and respective duration, and then intersect the paths, and decide what to keep. This approach revealed some issues in understanding how to compare the different lists resultant from each tree and how to decide which path to use on each Job, having such issue.

Weighted Processes

To simplify the resolution of the scheduling problem and have something that could be implemented, it was researched a proposal using weights, where is given more value to certain

characteristics, than others, which, in resume makes it able to decide which Process is best for a certain Operation and which Process from a set of Processes, “fighting” for the same machine, should use it. The following explanation of the algorithm, is a copy of what was written on an issue on git, dedicated to it, on the at following link: https://github.com/KnownUsername/FJSSP-Advanced-Manipulations/issues/5 .

Job Weight

As superficially mentioned before, it's being considered the possible time for a job to be completed. To achieve the weight, it's being used the following components: **average time** for a job to be completed, the **probability of that job end in a longer time** than average, and **how much more time, the job can last, than the average** (worst time - average time). The idea is to, when choosing a process, give less priority to the ones which correspondent jobs last longer, and differentiate the ones which has similar average time, but have a higher probability of ending later than that value.

Process Weight

There isn't much being considered on a process's weight, the idea is to use the weight from its job, together with its duration. The way to do such relation, is based on a percentage, where is expected to give more importance to the actual process time than the job's weight. This value is intended to be an adjustable threshold, starting on 60%-40%.

The resultant value is the one used to choose which process from a list where all use the same machine, is going to be assigned to respective machine. The variable is called *cedency*, the higher the value, the less priority it has, so less likely to be assigned.

New structures

To use such weights and intermediate values needed, its needed 2 new structures.

WeightedJob

```
{
  Job job;
  float jobWeight;
  List* weightedProcesses;
}
```

WeightedProcess

```
{
  char* jobIdentifier; // To keep id when using only structures' address
  int operationId; // To keep id when using only structures' address
  Process* process;
  int startTime; // Time when machine ends last process currently assigned*
  int referenceTime; // process->time + startTime
  float cedency; // percent * process->time + (1-percent) * job->jobWeight -- illustrative
}
```

this doesn't mean the Process will start right over, it's just a measure to decide the best Process on an Operation, as Processes are being scheduled

Although a Job has Operations, and an Operation has Processes, when mixing all Processes together, as will be explained on next chapter, it's loosed the identifier, such from Operation and Job. To deal with it, these identifiers are stored on this structure.

Algorithm Explanation

The idea is to start with gathering on a list, the best Process from each Job on Operation 1; check the machine from its process; find the Process on the list with lower *cedency*, and assign it to respective machine (schedule it).

After the insertion, each WeightedProcess using the machine that was assigned, is updated on the field *startTime* and consecutively *referenceTime*. This is meant to not assign the best Process from a Job's Operation, simply because it has the lower time to be completed, but also to have in mind, the waiting time that Operation will have, if use certain Process.

Another thing to do before schedule another Process, is to replace the one scheduled, with the best Process of the next Operation of same Job. This occurs, because it might be better to do a Process from next Operation number, than to do one from Operation 1, in this case.

Next iteration, we do the same, but now we would take the machine id from the second position of the list (which would be a Process from 2nd job), taking the same path as explained before, and on the next iteration use a Process from another Job, over and over. This makes it more likely to use different machines, once its used all jobs - end of the list - the iteration is restarted to first position.

Schedule of a Process

This action involves two operations: **store on a ScheduledOperation**, which is assigned to a ScheduledJob and **update the last time mark** used on the machine used on the Process intended to be stored. This last data is simply represented as follows :

```
{
  int machine;
  int timeUnits; // Last position corresponds to the units of time used
}
```

This information is needed, to keep track of how long a machine is being used, to validate if an Operation should use it.

Data Visualization

Output is stored in a csv text file and can be, then, visualized on a gantt map, using Python, generated with *plotly.express*. Here, we can check, an example of an output.

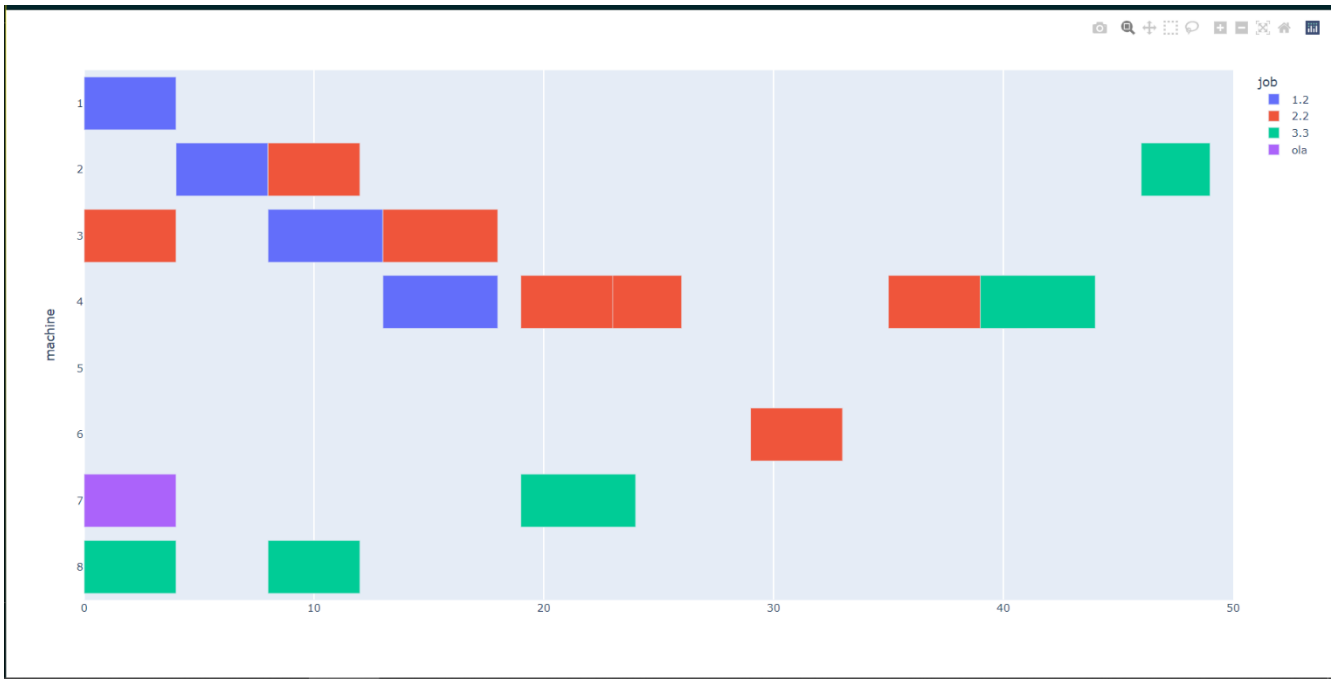

Figure 1 Map gantt generated from Python