FJSSP

Flexible Job Shop Scheduling Problem

Practical Work I

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 first approach of solving scheduling problems of FJSSP(*Flexible Job Shop Scheduling Problem)*. The missing part of problem's solution, will be developed on second practical work.

The implementation is made in C programming language, with the main objective to use linked lists.

João Rodrigues, 16928

LESI, IPCA
Advanced Data Structures
2021/2022

Index

Summary 2
Index 3
Contextualization 4
Problem in hands 4
Requirements 5
Entities 5
Data Structures 6
Structs 6
  Process 6
  ProcessList 7
  Operation 8
  OperationList 8
  Job 9
  JobProcess 9
Process Development 10
Data File Storage 10
  Load a file 11
  Save a file 11
Conclusion 11
Webography 12

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 manipulate linked lists, implying:

- Create lists
- Insert elements
- Edit elements
- Remove elements
- Search elements

The development of such project, allows students to understand the need to use dynamic data structures, when leading with data with variable amount of variables.

Problem in hands

It was provided a document with a problem of type FJSSP(*Flexible Job Shop Scheduling Problem*), which represents the scheduling of tasks of variable process plans, also known as *Jobs*. Each process plan, has a set of operations that must be done by order, and each can be completed by 1 or more machines, taking times that might not be the same. Each machine can only perform 1 operation at a time. Below, there's an example of a representation of a table, containing jobs and its respective operations, with the different machines and time usage, to perform them.

Process Plan	**Operation**						
	O 1	**O 2**	**O 3**	**O 4**	**O 5**	**O 6**	**O 7**
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			
$pr_{2,2}$	(1,3,5) [1,5,7]	(4,8) [5,4]	(4,6) [1,6]	(4,7,8) [4,4,7]	(4,6) [1,2]	(1,6,8) [5,6,4]	(4) [4]
$pr_{3,3}$	(2,3,8) [7,6,8]	(4,8) [7,7]	(3,5,7) [7,8,7]	(4,6) [7,8]	(1,2) [1,4]		
$pr_{4,2}$	(1,3,5) [4,3,7]	(2,8) [4,4]	(3,4,6,7) [4,5,6,7]	(5,6,8) [3,5,5]			
$pr_{5,1}$	(1) [3]	(2,4) [4,5]	(3,8) [4,4]	(5,6,8) [3,3,3]	(4,6) [5,4]		
$pr_{6,3}$	(1,2,3) [3,5,6]	(4,5) [7,8]	(3,6) [9,8]				
$pr_{7,2}$	(3,5,6) [4,5,4]	(4,7,8) [4,6,4]	(1,3,4,5) [3,3,4,5]	(4,6,8) [4,6,5]	(1,3) [3,3]		
$pr_{8,1}$	(1,2,6) [3,4,4]	(4,5,8) [6,5,4]	(3,7) [4,5]	(4,6) [4,6]	(7,8) [1,2]		

This problem is divided in two parts, where in one, it's approached a more low-level view of the problem, leading with only 1 job; and the other intends to use more than 1 job, managing their production order.

For first part, which is the one implemented on this project's document, the main objective is to define only 1 job and be able to manipulate operations. To do so, it's intended to use lists, as data structures.

For second part, as mentioned earlier, we have a bigger picture of the problem, using content approached on the first part, but now, giving a final solution, where is developed the use of multiple jobs, and a way to manage their production. On this part, the data structures may vary from linked lists, as others will be aborded, such as dictionaries.

On this document, it will only be talked about the first part, as it's the one implemented, so any reference to the word "project" is relative to it.

Requirements

The enunciation of the project asks for the follow:

1. Definition of a dynamic data structure, to represent a job with a finite set of n operations
2. Storage and reading of a text file, with a job representation
3. Insert of an Operation
4. Removal of a certain Operation
5. Changing a certain Operation
6. Determination of minimum time units' quantity, necessary to complete a job and listing of corresponding operations
7. Determination of maximum time units' quantity, necessary to complete a job and listing of corresponding operations

So, basically, it's meant to use lists, and CRUD operations with it. Adding to that, we must find a way to store information on a text file and find minimum and maximum time that costs to complete a job, and how the operations would be done (which machine at how much time).

Entities

A way to look at the project's problem, and group information, is to define entities, which can be represented as structures, in C, and with the table provided on the enunciation, the same as displayed earlier, it's easy to capture them. In this implementation, it was distinguished: **Job**, **Operation** and **Process**.

Data Structures

Since we need to use a dynamic data structure, is used lists to store data, since is the first data structure approached on class, as well as the unique, during the project's implementation. Arrays were also used, but as auxiliar to store string inputs.

Structs

As mentioned on last chapter, entities can be represented in C, as structs, so each entity has a representation in struct, resulting in 3. For each entity, there's associated a struct for lists of it, containing entity's structure and a pointer to next element, although list of Jobs is not used on this project. To end, there's also a structure made to obtain 2 values, from 2 functions, which stores the Job with 1 Process for each Operation , chosen accordingly with the function being for maximum or minimum time, which is present on Requirements' chapter on point 7 and 8. This way, results on the follow structures:

- Process
- ProcessList
- Operation
- OperationList
- Job
- JobList
- JobProcess

Process

Contains machine used on the Process and the time needed to complete a Process.

```
/// <summary>
/// Defines 1 process
/// </summary>
typedef struct Process{
    int machine;
    int time;
}Process;
```

On table, it's the representation of each red rectangle, where on top is the machine and on bottom, time.

Process Plan	Operation						
	O 1	O 2	O 3	O 4	O 5	O 6	O 7
$pr_{1,2}$	(1 3) [4 5]	(2 4) [4 5]	(3 5) [5 6]	4 5 6 7 8) 5 5 4 5 9]			

Figure 1 Values' correspondence to a Process structure

So, a Process, is a way to do an operation – a process.

ProcessList

Contains a struct Process and a pointer to another struct of type ProcessList.

```
/// <summary>
/// Defines a list of Processes
/// </summary>
typedef struct ProcessList {
    Process process;
    struct ProcessList* nextProcess;
}ProcessList;
```

Visualized on table as follows:

Process Plan	Operation						
	O 1	O 2	O 3	O 4	O 5	O 6	O 7
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			

Operation

An Operation is a task with an order associated to a Job. It's constituted by its order on a job and the list of possible Processes to do it.

```
/// <summary>
/// Defines 1 Operation
/// </summary>
typedef struct Operation {
    int opIdentifier;
    ProcessList *alternProcesses; // List of possible processes to complete an operation
}Operation;
```

Seen in the table as follows:

Process Plan	Operation						
	O 1	**O 2**	**O 3**	**O 4**	**O 5**	**O 6**	**O 7**
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			

OperationList

List implementation for Operations, follows same bases as ProcessList – 1 Process struct and 1 pointer to another struct of type OperationList.

```
/// <summary>
/// Defines a list of Operations
/// </summary>
typedef struct OperationList {
    Operation operation;
    struct OperationList* nextOperation;
}OperationList;
```

On table, this is a list with all Operations of a job.

Process Plan	Operation						
	O 1	**O 2**	**O 3**	**O 4**	**O 5**	**O 6**	**O 7**
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			

Job

A job is a set of tasks (Operations), that are done on an order (each Operation has an order, as mentioned on its chapter). This is the higher level of entities and contains its identifier and correspondent operations.

```
/// <summary>
/// Defines 1 job
/// </summary>
typedef struct Job {
    char* jobIdentifier;
    OperationList* operations; // List of operations to complete a job
}Job;
```

From table, here's a representation of 1 job. A process plan is the same as a job.

Process Plan	Operation						
	O 1	**O 2**	**O 3**	**O 4**	**O 5**	**O 6**	**O 7**
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			
$pr_{2,2}$	(1,3,5) [1,5,7]	(4,8) [5,4]	(4,6) [1,6]	(4,7,8) [4,4,7]	(4,6) [1,2]	(1,6,8) [5,6,4]	(4) [4]

JobProcess

Contains a Job, which instead of having a list of processes on each Operation, it's meant to only contain 1, that satisfies a certain criteria and total time to do such Job (sum of each Process's time on each Operation). For this project, is used to store minimum and maximum time path to do a Job.

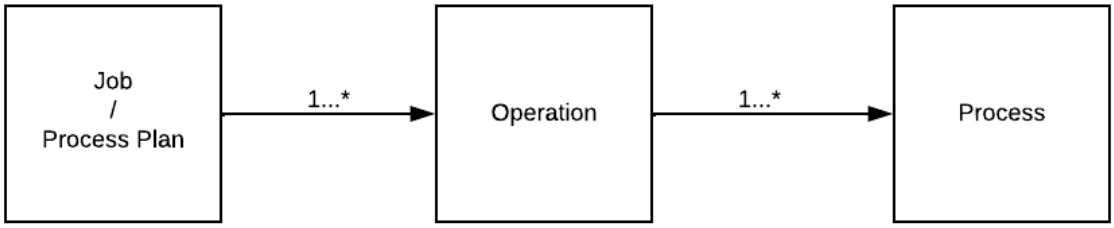
```
/// <summary>
/// Defines a Job with a Process established for each Operation,
/// instead of having all possibilities.
///
/// It's also added the full duration of the Job.
/// </summary>
typedef struct JobProcess{
    Job job;
    int fullDuration;
}JobProcess;
```

As this struct was developed just to store insights from data, there's no representation on the table.

Process Development

First step was to create 3 header files, 1 for each entity, where was defined respective structs (for entity and list of entities). It was also added a *main.c*, including already all headers, to posteriorly test functions leading with its structs.

Then, followed a perspective of the lower level / more granular part of the project, to the higher one - basically start with the components, to reach the final product. This, plus project's requirements, which kind of interconnected with each other. Having this in mind, the order of development establishes as: Processes -> Operations -> Job, as the right ones, depend from the left.



A Job has Operations, and an Operation has Processes.

Data File Storage

To store data in files, it was used csv files, where lines, besides the first one, which is reserved for the name of columns, represents a record with the format: Id of job, id of operation, machine of process and time to do the process. Example of representation:

```
process_plan,operation,machine,time
1.2,1,1,4
1.2,1,3,5
1.2,2,2,4
1.2,2,4,5
```

Load a file

On loading file, it's used *fscanf()*, applying regular expressions.

- **Header** – For columns' names, it's ignored its values and the regular expression is: **%[^\,],%[^\,],%[^\,]\,%[^\n]\n**, consisting of reading each string between commas and then last value is till \n, when it's changed line.

- **Values** – In terms of values, regular expression used is **%[^\,]\,%d\,%d\,%d\n** , reading firstly a string (job's identifier is a string) and integer numbers, on follow fields between commas and till \n (new line).

Save a file

Saving a file is more straightforward, where is simply needed to pass values of each record, through *fprintf()*.

- **Header** – On header, where columns' names reside, the content is always the same, to keep its consistency leading to the follow format:

```
// Header
fprintf(fp, "%s,%s,%s,%s\n", "process_plan", "operation", "machine", "time");
```

- **Values** – For values is, as said on sub-chapter's introduction, just passing values, leaving to this:

```
// Job id | Operation id | machine | process time
fprintf(fp, "%s,%d,%d,%d\n", job.jobIdentifier, operationList->operation.opIdentifier,
        processList->process.machine, processList->process.time);
```

Conclusion

With this project, is possible to understand the need to use dynamic data structures, as it helps on storing only the memory needed and not more or less than that. Now, this doesn't mean that dynamic is better than static, it's all about context, if there's a fixed size to an amount, an array may work, and its navigation to a specific position is more direct, as is index based, while on a list, it must navigate through the elements behind, to reach it.

In terms of implementation, the project's code is able to realize all functionalities proposed on Requirement's chapter and respective documentation is available on Git.

To consult documentation and code, GitHub link: https://github.com/KnownUsername/FJSSP-Basic-Manipulations

Webography

https://stackoverflow.com/questions/24295820/doxygen-isnt-generating-documentation-for-source-files - To fix documentation generation, in terms of files not being included

https://www.fujipress.jp/ijat/au/ijate001300030389/ - To understand what a FJSSP is