# 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.

# Index

# Figures’ 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 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,

| **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] | | |

*Figure 1 - Example of a table of a problem FJSSP, provided by enunciation*

containing jobs and its respective operations, with the different machines and time usage, to perform them.

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
8. Determination of average time units' quantity necessary to complete an operation, considering all alternative possibilities.

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 dynamic 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;
```

*Figure 2 - Implementation of structure Process*

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

| Process Plan | Operation | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 1 | 0 2 | 0 3 | 0 4 | 0 5 | 0 6 | 0 7 |
| $pr_{1,2}$ | (1,3)<br>[4,5] | (2,4)<br>[4,5] | (3,5)<br>[5,6] | (4,5,6,7,8)<br>[5,5,4,5,9] | | | |

*Figure 3 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;
```

*Figure 4- Implementation of structure ProcessList*

Visualized on table as follows:

| Process Plan | Operation | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 1 | 0 2 | 0 3 | 0 4 | 0 5 | 0 6 | 0 7 |
| $pr_{1,2}$ | (1,3)<br>[4,5] | (2,4)<br>[4,5] | (3,5)<br>[5,6] | (4,5,6,7,8)<br>[5,5,4,5,9] | | | |

*Figure 5 - Values' correspondence to a ProcessList structure*

## 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.

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

*Figure 6- Implementation of structure 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)<br>[4,5] | (2,4)<br>[4,5] | (3,5)<br>[5,6] | (4,5,6,7,8)<br>[5,5,4,5,9] | | | |

*Figure 7 - Values' correspondence to an Operation structure*

## OperationList

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

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

*Figure 8- Implementation of structure 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)<br>[4,5] | (2,4)<br>[4,5] | (3,5)<br>[5,6] | (4,5,6,7,8)<br>[5,5,4,5,9] | | | |

*Figure 9 - Values' correspondence to an OperationList structure*

## 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;
```

*Figure 10- Implementation of structure Job*

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

| Process Plan | Operation | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 1 | 0 2 | 0 3 | 0 4 | 0 5 | 0 6 | 0 7 |
| $pr_{1,2}$ | (1,3)<br>[4,5] | (2,4)<br>[4,5] | (3,5)<br>[5,6] | (4,5,6,7,8)<br>[5,5,4,5,9] | | | |
| $pr_{2,2}$ | (1,3,5)<br>[1,5,7] | (4,8)<br>[5,4] | (4,6)<br>[1,6] | (4,7,8)<br>[4,4,7] | (4,6)<br>[1,2] | (1,6,8)<br>[5,6,4] | (4)<br>[4] |

*Figure 11 - Values' correspondence to a Job structure*

## 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;
```

*Figure 12- Implementation of structure 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.

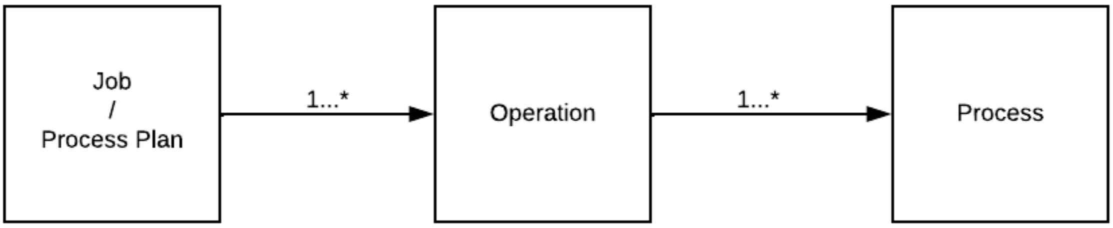


*Figure 13- Representation of connection between the 3 entities*

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
```

*Figure 14 - Part of exapmle csv file to manipulate project's data*

### 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");
```

*Figure 15 - Implementation of reading values from file's header*

- **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);
```

*Figure 16 - Implementation of reading values from file's body/values*

## Tests

As the project was being implemented, the functions were being tested on a file that is now called tests.c . At moment, there's an interface program, to allow the use of those functions freely on a console, which is main.c .

## Program Start

This paragraph is meant to give a context on where the tests are done, to show on this report.

First of all, there must be a Job instance, as the project is meant to lead with a single Job, so, first step is to ask to create 1, having source from a file, or creating from scratch. This is what it looks like, the first menu:

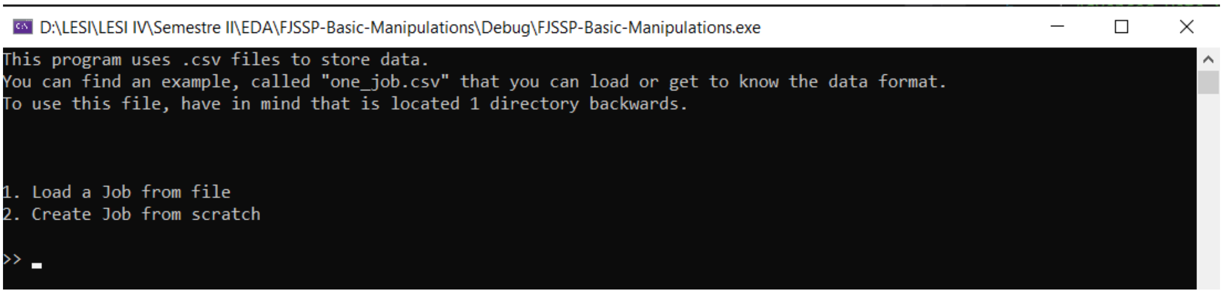


*Figure 17 - Starting menu*

To help the user, it's given an example file to use the operations, revealing its file name and directory. The option is chosen with numbers.

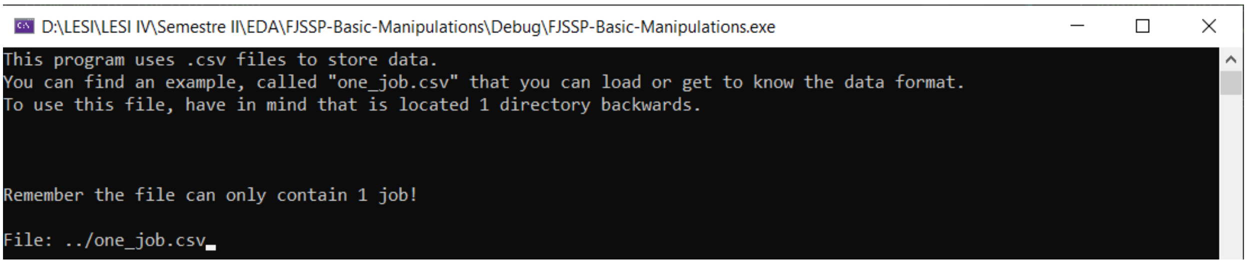


*Figure 18 - Input of a file to load*

Which, on success leads us to the main menu. This is where is given the operations we can use to manipulate our data.

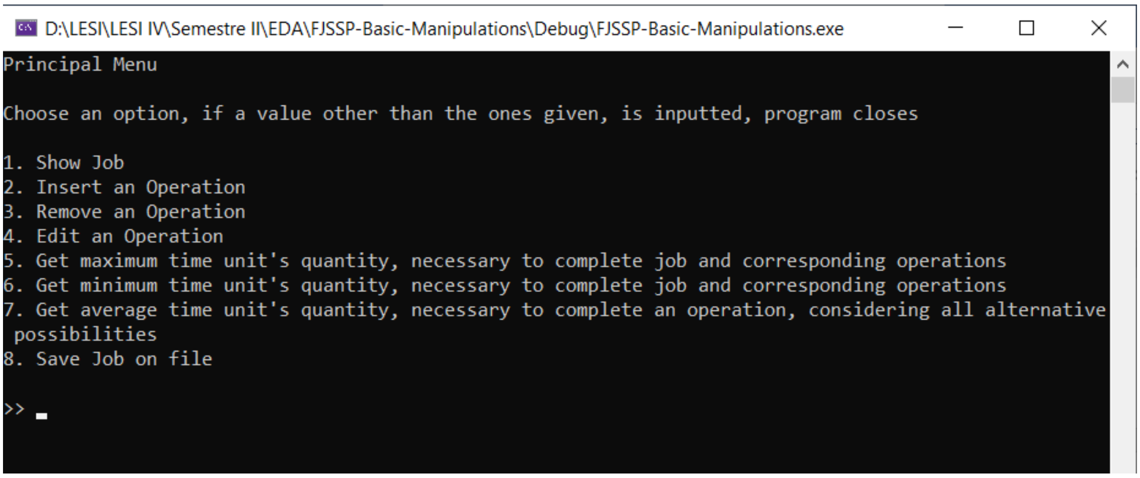


*Figure 19 - Demonstration of main menu*

## Load a file

By choosing “Show Job”, we can see that data chosen from previous chapter was received.

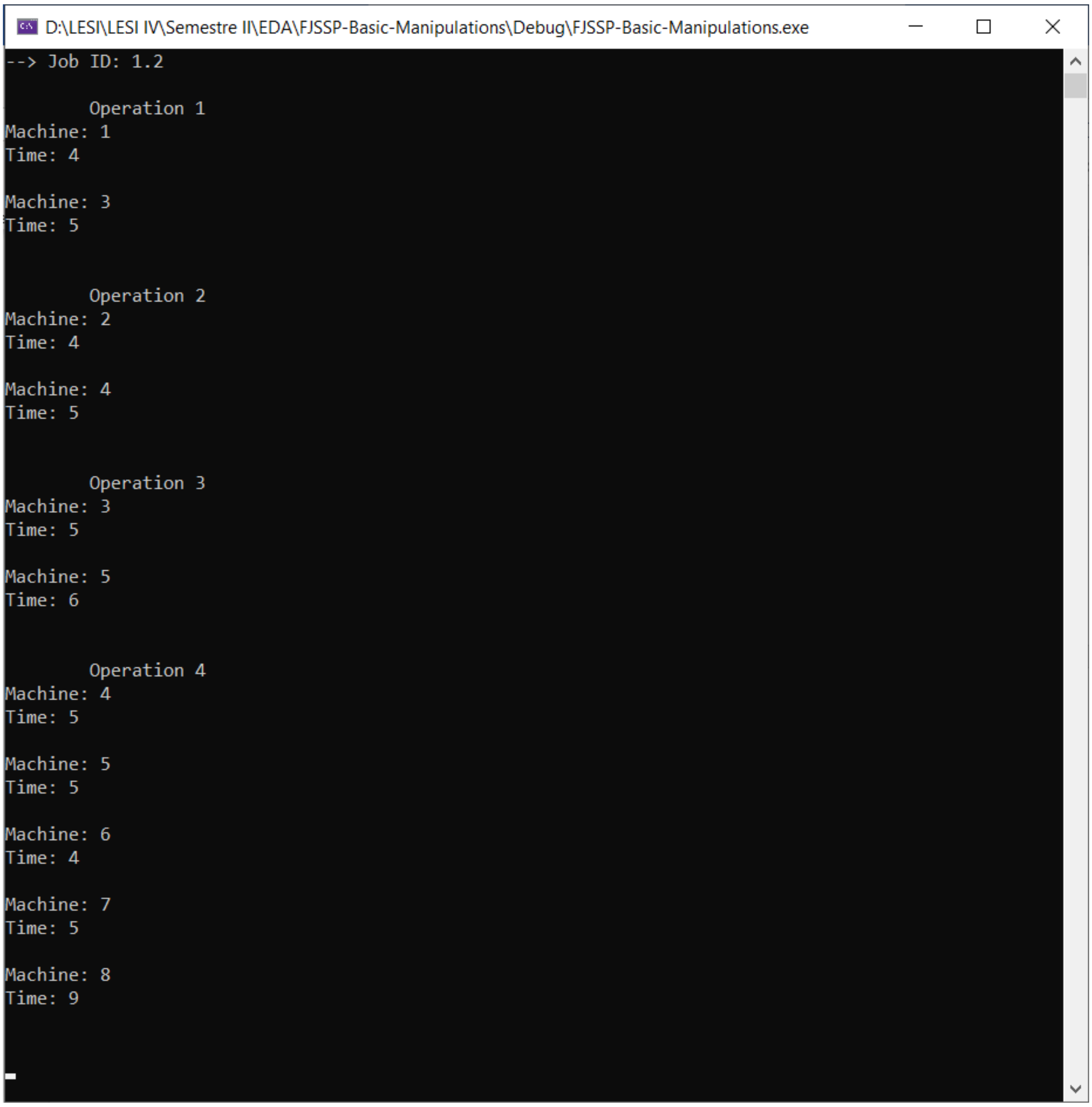


*Figure 20 - Demonstration of showing a job's values from loading a file*

Which matches the original file, being it:

```
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
1.2,2,2,4
1.2,3,3,5
1.2,3,5,6
1.2,4,4,5
1.2,4,5,5
1.2,4,6,4
1.2,4,7,5
1.2,4,8,9
```

*Figure 21 Csv file that serves as example for the project*

The understanding of the values can be found previously on the chapter "Data File Storage". As we can see, the values match, so it successfully imported data, and assigned it to a Job.

**Inexistant files**

When is inserted a file that doesn't exists, the program tells that to the user, and by pressing enter, he may try to put another.

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-Manipulations\Debug\FJSSP-Basic-Manipulations.exe
This program uses .csv files to store data.
You can find an example, called "one_job.csv" that you can load or get to know the data format.
To use this file, have in mind that is located 1 directory backwards.



Remember the file can only contain 1 job!

File: inexistent
File does not exist.
```

*Figure 22 - Demonstration of inputting an inexistant file*

When creating from scratch, there isn't much to tell, it's simply needed to give an identifier and it will open the main menu.

From now on, we'll be exploring the program's functionalities, and its behaviors based on the file's input.

## Show a Job

As seen before, this function works, having shown the Job created from the file “one_job.csv”. The other case we might want to test is, if the Job has no values.

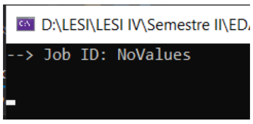


*Figure 23 - Demonstration of showing a Job with no values*

It’s concludable, that with no values, the function doesn’t give an error, and simply leaves blank, as there’s no values, something we would expect to happen on similar scenarios. On an implementation point of view, this means that NULL lists, have no problem on this function.

Note that for this example, we left out the data from the file, and started a new Job from scratch, with the identifier “NoValues”.

## Insert an Operation

**Regular Case**

To keep the implementation of this interface simple, when an Operation is inserted, the user can only choose its identifier; and to have Processes associated to it, it must edit the Operation, adding them up.

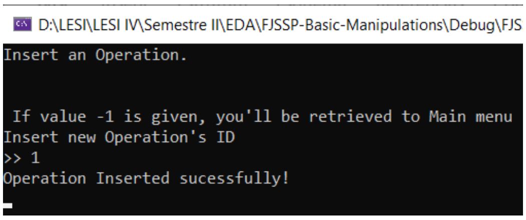


*Figure 24 – Demonstration of inserting an Operation, simply inputting its identifier*

We now go check if was added, on the option to show.

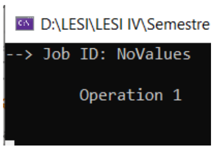


*Figure 25 - Proving the Operation was inserted on the Job*

And there it is, the new Operation, with no Processes associated.

**Using a repeated Identifier**

We now, get back to the Job created from a file. Recalling the Job data.

```
D:\LESI\LESI IV\Semestre
--> Job ID: 1.2

        Operation 1
Machine: 1
Time: 4

Machine: 3
Time: 5


        Operation 2
Machine: 2
Time: 4

Machine: 4
Time: 5


        Operation 3
Machine: 3
Time: 5

Machine: 5
Time: 6


        Operation 4
Machine: 4
Time: 5

Machine: 5
Time: 5

Machine: 6
Time: 4

Machine: 7
Time: 5

Machine: 8
Time: 9
```

*Figure 26 - Data from Job original from "one_job.csv"*

So now, creating an Operation with an identifier already being used, results in follows:

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-Manipulations\Debug\FJSSP-Basic-Manipulations.exe
Insert an Operation.


 If value -1 is given, you'll be retrieved to Main menu
Insert new Operation's ID
>> 2

Id is already taken!
```

*Figure 27 - Behavior of inserting an Operation, using an identifier that is already being used*

The user gets to know that the identifier is being used, which means the Operation is not added, and there’s no overlap.

### Use an invalid input

The identifier of an Operation is labelled as an integer, and is meaningful for values higher than 0. 0 is not being allowed because of using *atoi()*, which retrieves 0 on values that can't be converted from string to number.

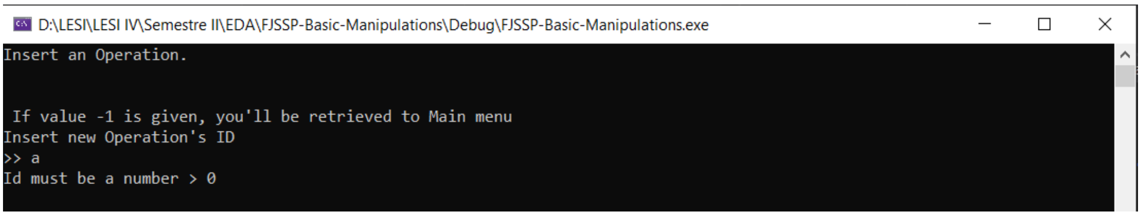


*Figure 28 - Example of an invalid input*

## Remove an Operation

### Regular case

To remove an Operation, it's simply needed to pass its identifier.

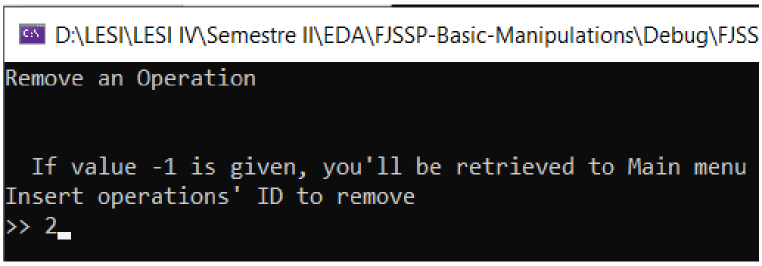


*Figure 29 - Input of removing an Operation*

As there's an Operation with given identifier, the program claims it was removed.

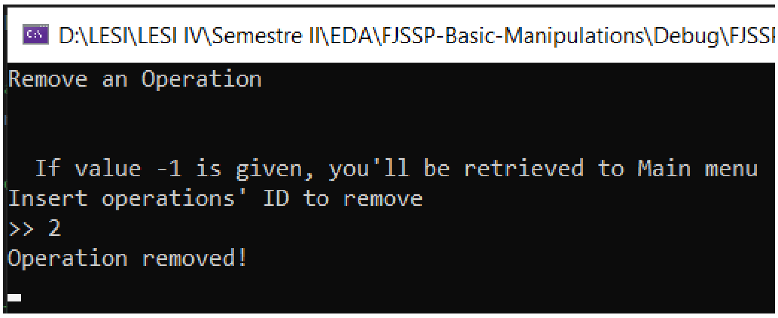


*Figure 30 - Answer from removal, on success*

Verifying on Show().

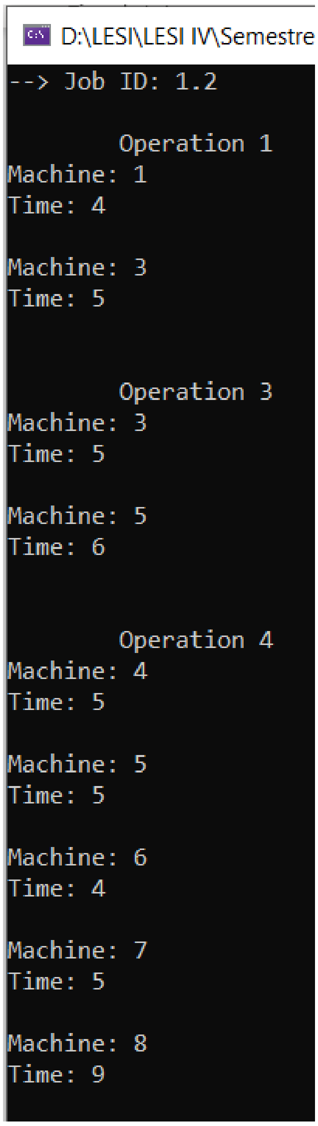


*Figure 31 - Preview of Job after removal of Operation 2*

Operation 2 is no longer present.

**Inexistent identifier**

By passing an identifier that doesn't exist, the program retrieves the inexistence of such identifier, implying the inexecution of a removal.

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-Manipulations\Debug\FJSSF
Remove an Operation


  If value -1 is given, you'll be retrieved to Main menu
Insert operations' ID to remove
>> 5

No Operation with ID = 5, was found
```

*Figure 32 - Behavior of removal when ID isn't present on list*

## Edit an Operation

To edit an Operation, it must be given and identifier that exists. By using an inexistant, it's given a warning as in the previous test.

To help the user pick an identifier, it's shown all Operation available on current Job.

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-Manipulations\Debu
        Operation 1
Machine: 1
Time: 4

Machine: 3
Time: 5


        Operation 3
Machine: 3
Time: 5

Machine: 5
Time: 6


        Operation 4
Machine: 4
Time: 5

Machine: 5
Time: 5

Machine: 6
Time: 4

Machine: 7
Time: 5

Machine: 8
Time: 9


If you submit -1, you'll be redirected to Main menu

Operation ID to edit: 4
```

*Figure 33 - Input of an Operations' identifier to edit*

On success, we're redirected to a new menu, residing all options to edit an Operation.

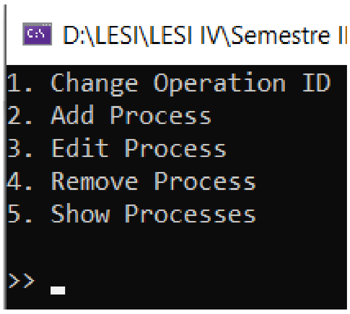


*Figure 34 - Menu to edit a chosen Operation*

## Change Identifier of an Operation

**Regular case**

Choosing option 1, we're presented the identifier currently being used, so there's no need to memorize it. On this example, we changed the identifier to 5, a value unused.

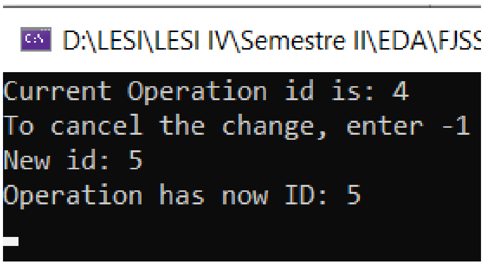


*Figure 35 - Process of changing an Operation's identifier*

Confirming modification.

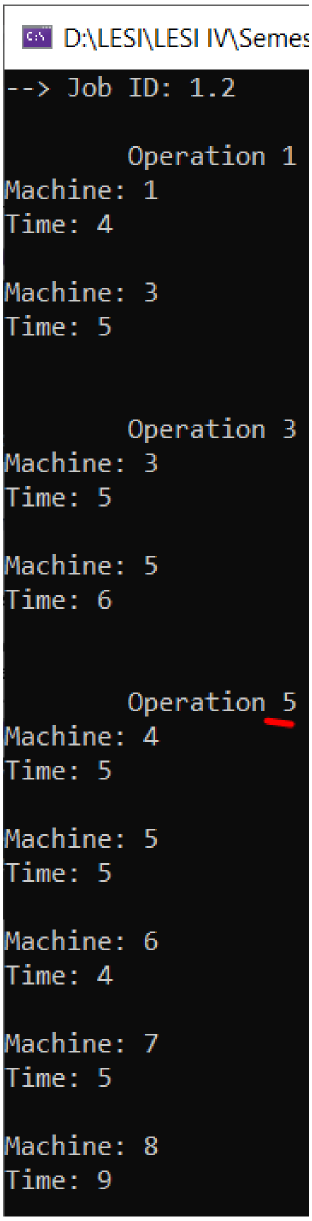


*Figure 36 - Listing values to confirm the Operation's identifier change*

**Identifier already being used**

Choosing an identifier that already exists, doesn't apply the change, warning for the problem

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-Manipulations\Debug\FJSSP-Basic-Manipulations.exe
Current Operation id is: 5
To cancel the change, enter -1
New id: 1

An Operation with ID = 1, was already created. Please choose a different value!
```

*Figure 37 - Behavior on changing an identifier to an already existent*

**Invalid Identifier**

An identifier on an Operation, can only be integer numbers higher than 0. By writing a string, the problem is exposed to the user, not assigning new identifier.

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSI
Current Operation id is: 5
To cancel the change, enter -1
New id: string
Id must be a number > 0
```

*Figure 38 - Warning when trying to use an invalid identifier*

## Add Process to a list of Processes

**Regular case**

Choosing option 2, we're able to input the values for machine and time. The values are verified, after inputted both.

```
D:\LESI\LESI IV\Semestre II\EDA\FJSSP-Basic-I
Add a Process


 Insert -1 to retrieve to Main Menu
Machine: 2
Time: 1
Process added with success!
```

*Figure 39 - Creation of a Process*

The values inputted are valid, so we got success on the operation, making current job, as this:

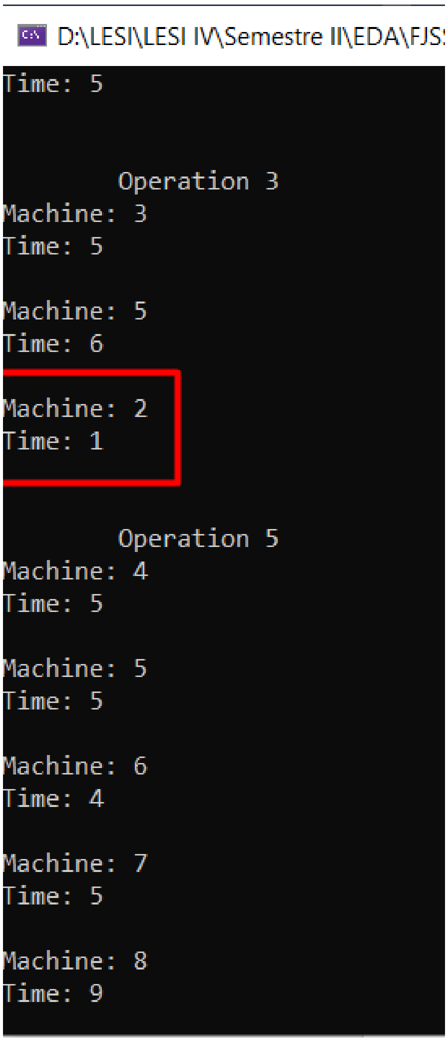


*Figure 40 - Listing of Job after adding a Process*

The Process was inserted on the Operation 3, that's why it's displayed there.

**Other cases**

Both situations for input of already existent or invalid identifier, are also considered on Processes, same way as it is on Operations, so it's redundant to repeat the information.

## Edit a Process

Choosing option 3, it’s displayed the Processes on chosen Operation, so the user know his possibilities. Using Operation 5, we’re going to edit Process from Machine 8.

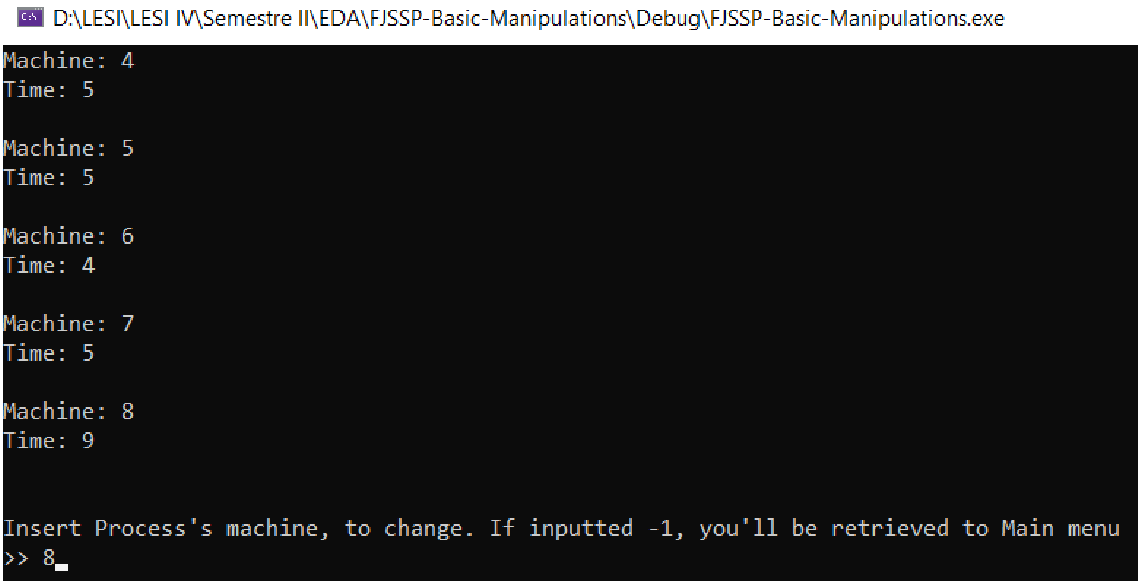


*Figure 41 - Choose of a machine's Process to change Process*

Submitting, it’s needed to pick the attribute we want to change on a Process, and a Process only has a machine and time.

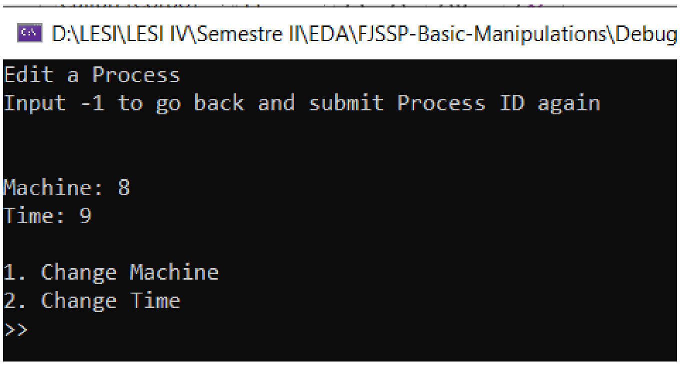


*Figure 42 - Menu to pick between changing machine or time*

For this test, it was inputted to change time to 10.

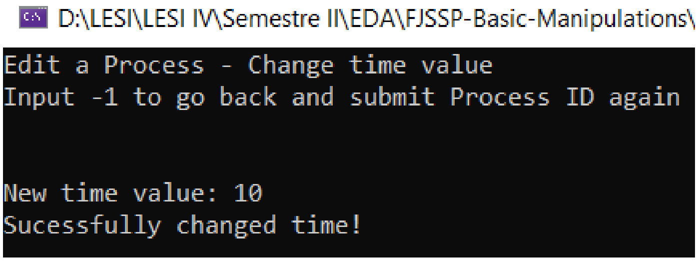


*Figure 43 – Successful change of time of a Process*

The result replies with success.

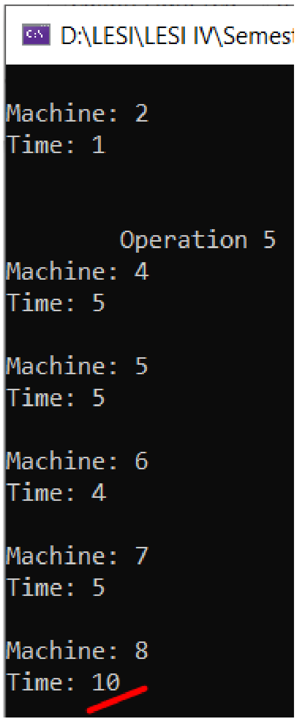


*Figure 44 - Demonstration of a Process updated after being changed*

**Inputs’ Validation**

For time, is checked if it’s a number with value >= 0 and for machine, it’s the same validation as in introducing a new machine.

## Remove a Process from a list

Choosing option 4, it's displayed Processes of chosen Operation. On this example, it was chosen Operation 3.

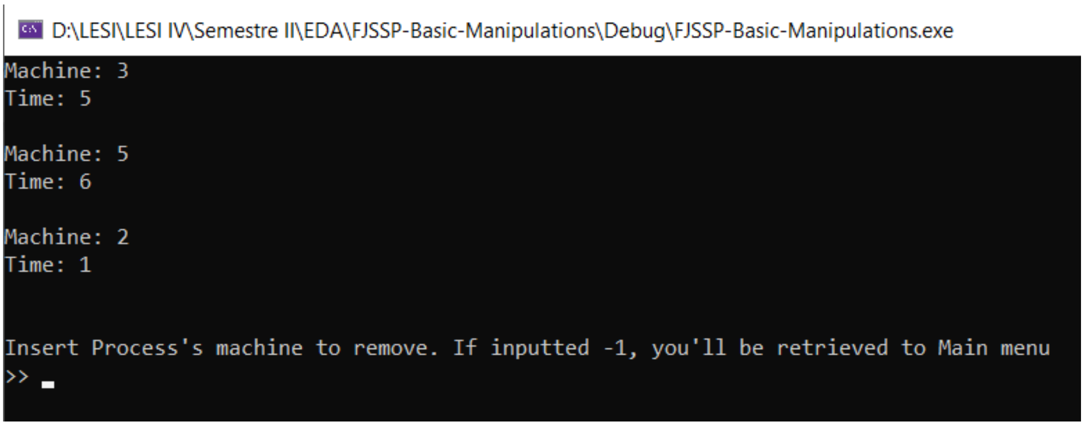


*Figure 45 - Display to remove a Process*

The behavior is the same as removing an Operation, as documented at chapter "*Remove an Operation*" – same verifications (existence of machine to remove) and same format of messages on success or unsuccess on removing.

After removed the Process on machine 2, here's the updated list:

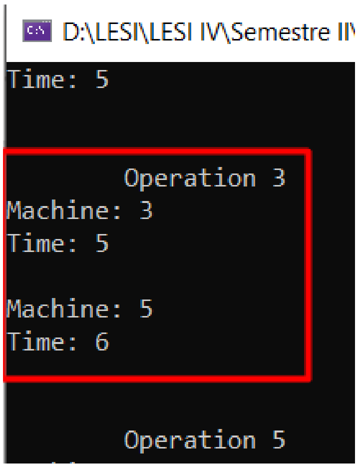


*Figure 46 - Display of Operation 3, after removing its Process on machine 2*

Process on machine 2 is gone, leaving the others.

## Print Processes from a list

Choosing option 5, it simply displays Processes on Operation. Using Operation 5, here's the output.

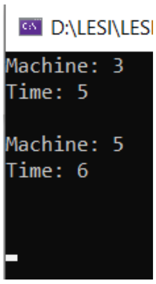


*Figure 47 - Demonstration of Showing a list of Processes with values*

**Empty List**

On empty list, displays the same as a Job without Operations – nothing. For demonstration, will be used a new Operation with no Processes.

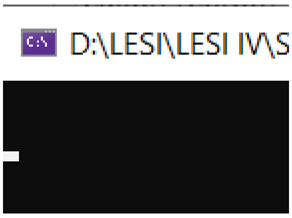


*Figure 48 - Demonstration of a list of Processes empty*

Pure blank and no problem on leading with NULL lists.

## Get maximum time unit's quantity, necessary to complete job and corresponding operations

For each Operation is being chosen the first machine found with higher time and the sum of all times, as asked.

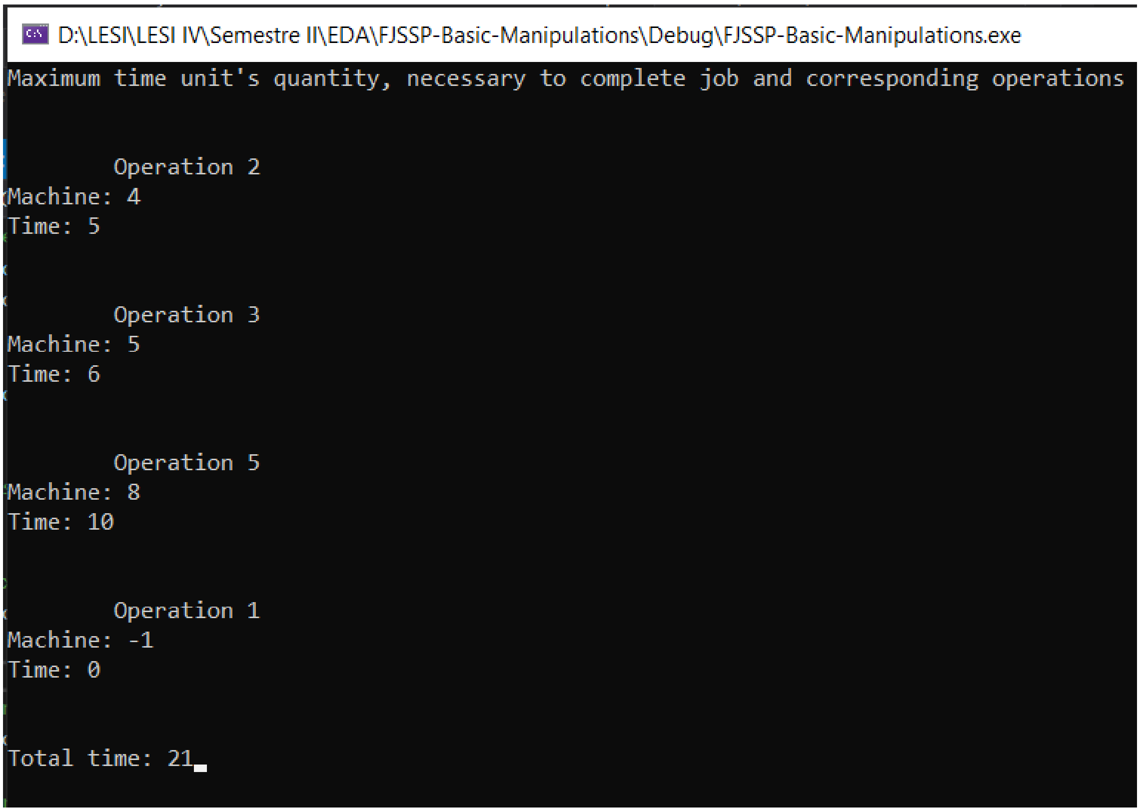


*Figure 49 - Result of  maximum times for each Operation and the sum of all*

On Operation 1, there's a special case, where was introduced machine as -1 and time at 0. This happens, because the Operation has no Processes, so, this is the way used to treat such cases, using an identifier that can't be introduced usually – so the user knows that the Operation has no values, and a time 0 to not influence the sum of values.

## Get minimum time unit's quantity, necessary to complete job and corresponding operations

For each Operation is being chosen the first machine found with higher time and the sum of all times, as asked.

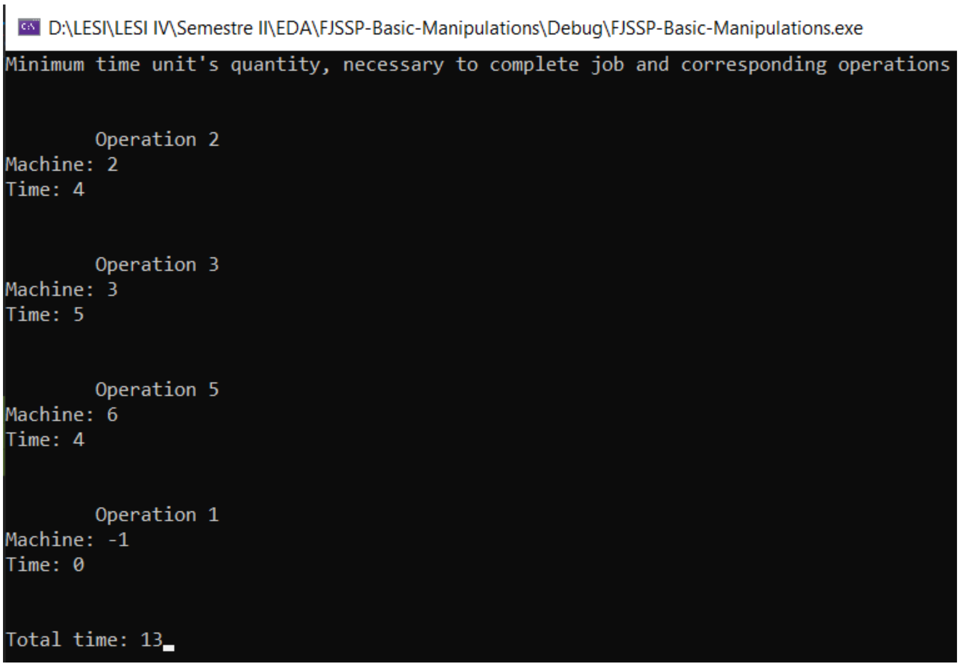


*Figure 50 - - Result of minimum times for each Operation and the sum of all*

For Operations with no Processes, the problem is dealt as in the previous feature, for reaching maximum time, where it's returned a Process with machine = -1 and a time of 0.

## Get average time unit's quantity, necessary to complete an operation, considering all alternative possibilities

Calling option for average of our Job, we get the follow:

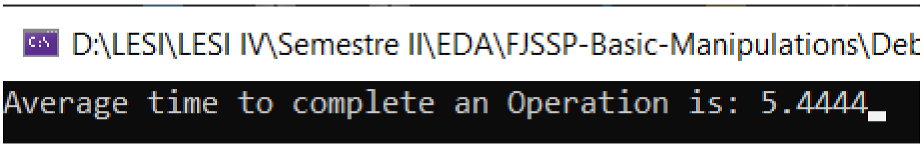


*Figure 51- Result of average time to complete an Operation*

To confirm the average returned, we can calculate to assure it.

**Values**

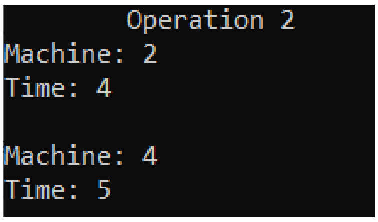


*Figure 53 - Processes on Operation 2*

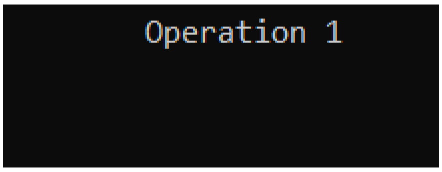


*Figure 52- Processes on Operation 1*

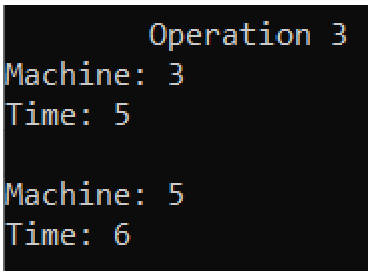


*Figure 54- Processes on Operation 3*

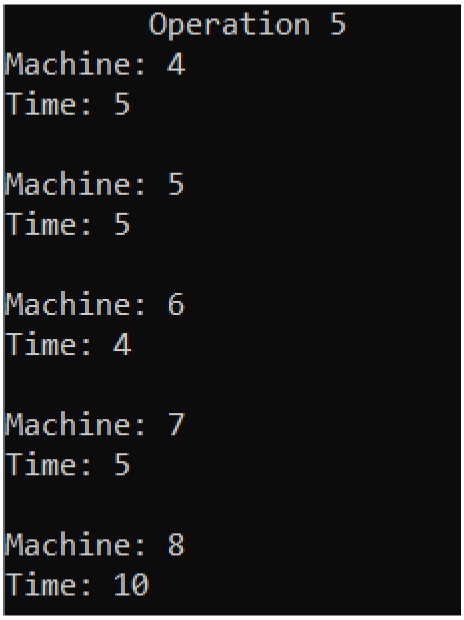


*Figure 55 - Processes on Operation 5*

**Calculus**

Average = Sum / N (number of elements)

Sum = (4+5) + (5+6) + (5+5+4+5+10) = 49

N = 9 (counting all processes)

Average = 49 / 9 = **5.4(4)**

**Conclusion**

As the result from handmade calculus is the same as the one given from the program, the average is being calculated correctly.

## Save Job on a file

To save a Job on a file, all we need to do, is to give a name to our new file.

```
D:\LESI\LESI IV\Semestre II\EDA\
Filename: tests_file.csv
Sucess on saving to file!
```

*Figure 56 - A file being successfully created*

Which leads to the generation of the follow text file.

```
process_plan,operation,machine,time
1.2,2,2,4
1.2,2,4,5
1.2,3,3,5
1.2,3,5,6
1.2,5,4,5
1.2,5,5,5
1.2,5,6,4
1.2,5,7,5
1.2,5,8,10

```

*Figure 57 - Display of generated file*

# 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](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