MANUAL

1. Introduction.

The scheduler is able to get a task schedule taking into account different machines, formulas, orders and products. The schedule is able to deal with very difficult and different production processes, it is able to decide which is the best way to make a product if two alternatives for a task are present (multistage processes). Moreover, it is able to give a worker schedule assigning them to different machines and shifts. The user can cancel the module of workers or the maintenance/cleaning tasks (cip) if need it.

The scheduler has two modes of functioning, standalone and API. These two modes are similar and only they differ in the initialization process. The standalone mode gives a graphic representation of the solution, on the other hand, the API gives the ability to run the scheduler on a server and call it through a HTTP request. This API architecture only allows to communicate to the scheduler through a single JSON file.

The scheduler is based on a constraint/optimization satisfaction solver relying on high pre- and post-processing. On top of that, the scheduler performs reinforcement learning in order to achieve a better solution adapting to the different plants configurations.

2 . Inputs and outputs of the scheduler.

As said the scheduler will communicate with an input and output JSON file.

2.1 Input File:

It is going to have a single input file. This file consists in 7 different entries for the parameter definition of the scheduler. First, the problem_info entry, then, 3 entries regarding the plant information (formulas, machines and shifts). And finally 3 more entries for the definition of the problem (machine_states, workers and orders).

Next, each entry will be discussed in detail.

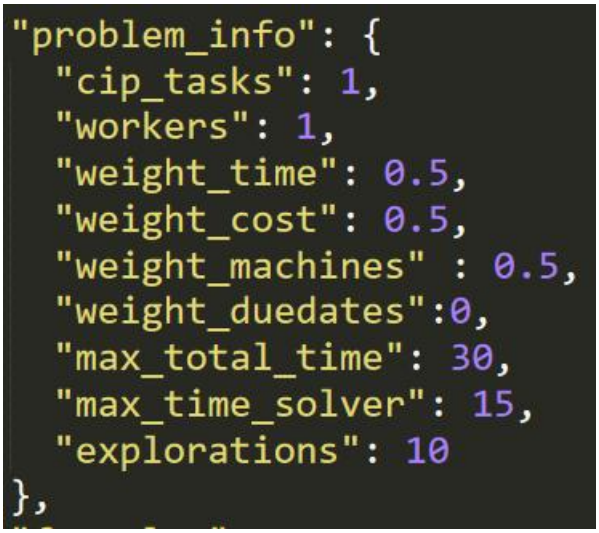
- problem_info:

It defines the configuration parameters of the scheduler and consists on the following:

- cip_tasks: If cip tasks are required.
- workers: if the worker scheduler is needed.

- weight_time, weight_cost, weight_machines i weight_duedates: This are the user preferences for the solution. The user can prioritize the makespan, cost, number of machines and the delay of the due dates with the weights respectively. Note 1: the scheduler normalizes this weights internally. Note 2: **It is very advisable that the last weight is set to a low number** (of the order of 0.01) even though this parameter wants to be prioritized.
- max_total_time: Approximate maximum time in seconds that the scheduler is given for achieving the most optimal solution.
- max_time_solver: Maximum time in seconds let to the solver for each solution.
- explorations: Quantity of solutions searched before updating the best solution. This parameter can be set to 5-15. Keep in mind that this parameter affects directly to the efficiency of the scheduler and depends on the size of the problem, max_time, and max_time_solver. If the scheduler needs a lot of time for each solution (complex and big problem) this parameter is better to keep it low, as the scheduler will pick a better solution faster, converging to a solution faster, with the throwback of less exploration done and the risk to not follow the best path.

An example of a problem_info entry:



- formulas:

This entry will have the information of how each product is made. Each element is a stage on the product production, so it will be a task needed to complete the product. Each stage has the following attributes:

- formula_id: formula unique identification, one for each step and product.
- product_id: product unique identification.

- step: integer that represents the order that has to follow the production in order to complete the product.
- step_pre: Which is the previous step needed in order to initiate this step. **This entry is a list.**
- machine_id: machine identification.
- Type: There is three type of possibilities:
  1. Time and cost needed for the step are affected by the quantity produced (type 1).
  2. Time and cost needed for the step is **not** affected by the quantity produced (type 2).
  3. The stage is parameterized as a ration(Kg/h) (type3). If this type is selected the processing_delay has to be set to 1 and the cost entry defines the cost for unit of time.
- max_output: maximum output that the machine can produce.
- processing_delay: time needed to perform this step.
- cost: cost for this stage (water, electric energy).

Example of a formula for a product:

```
"formulas":
[
  {
    "formula_id": 1,
    "product_id": 1,
    "step": 1,
    "stp_pre": [0],
    "machine_id": 1,
    "type": 1,
    "max_output": 50,
    "processing_delay": 4,
    "cost": 2
  },
  {
    "formula_id": 2,
    "product_id": 1,
    "step": 2,
    "stp_pre": [1],
    "machine_id": 2,
    "type": 1,
    "max_output": 25,
    "processing_delay": 4,
    "cost": 1
  },
```

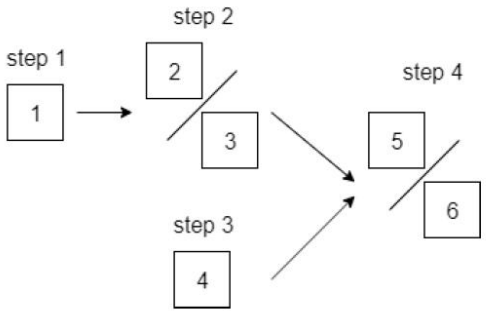
```
{
  "formula_id": 3,
  "product_id": 1,
  "step": 2,
  "stp_pre": [1],
  "machine_id": 3,
  "type": 1,
  "max_output": 25,
  "processing_delay": 2,
  "cost": 2
},
{
  "formula_id": 4,
  "product_id": 1,
  "step": 3,
  "stp_pre": [0],
  "machine_id": 4,
  "type": 1,
  "max_output": 100,
  "processing_delay": 10,
  "cost": 2
},
```

If there is more than one option to perform a stage it is identified with a different formula_id but the same step (multistage step).

To clarify how to encode a given complex formula an example is given:

Fo

fo_id	step	stp_pre	ma_id
1	1	[0]	1
2	2	[1]	2
3	2	[1]	3
4	3	[0]	4
5	4	[2,3]	5
6	4	[2,3]	6



You can see on the right the production process and on the left, how the particular encoding of the steps are correctly done. Note that there is branching and step 2 and 4 are multistage steps.

For more information go to the master thesis attached.

machines:

Information of the machines. Each element is a machine and it has the following attributes:

- machine_id: machine identification.
- clean_delay: Time needed for the maintenance/cleaning task (cip).
- clean_cost: Cost associated to the cip task
- clean_every: After how many working time the machine has to perform a cip task.

Example of two elements:

```
 "machines":
 [
{
 "machine_id":1,
 "clean_delay":4,
 "clean_cost":100,
 "clean_every":24
},
{
 "machine_id":2,
 "clean_delay":4,
 "clean_cost":50,
 "clean_every":24
},
```

shifts:

Here the worker shifts are defined. Even if the option ‘workers’ from the problem_info is set to 0, this entry must be present and, it must at least have an entry that is: [0]. An entry of shifts has the following architecture:

- shift_id: shift identification.
- day_of_the_week: day of the week. 1-7.
- shift_start: shift starting time.
- shift_end: shift ending time.

Example of two entries of shifts:

```
"shifts":
[
{
 "shift_id":1,
 "day_of_the_week":1,
 "shift_start": 8,
 "shift_end": 16
},
{
 "shift_id":2,
 "day_of_the_week":2,
 "shift_start": 8,
 "shift_end": 16
},
```

machine_state:

Entry that shows the availability of each machine. 1 means that it is available and 0 that is not.
The structure is the following:

```
 "machines_states":
{
"available":{
 "1":1,
 "2":1,
 "3":1,
 "4":1,
 "5":1,
 "6":1,
 "7":1,
 "8":1,
 "9":0
}
```

workers:

Entry that defines the workers. Each worker has a list associated with two elements. The first one is the availability, it has the same format as the availability of the machines and it is used to identify if a worker is ill or on vacation (general availability). The second element is a list of shifts identifications, this list show the shifts when the worker is available (particular availability).

```
"workers":
  {
    "available":{
  "1":[1,[1,2,3,4,5]],
  "2":[1,[1,2,3,4,5]],
  "3":[0,[1,2,3,4,5]],
  "4":[1,[1,2,3,4,5]],
  "5":[1,[1,2,3,4,5]],
  "6":[1,[1,2,3,4,5]],
  "7":[1,[1,2,3,5]],
  "8":[1,[1,3,4,5]]
  }
},
```

If the option ‘workers’ is set to 0, the procedure is the same as shifts. This entry must to be there and is has to have an entry [0] at minimum. This cases can be fully seen with the input example file test2.json.

Orders.json:

Entry where the orders that must be fulfilled are defined. It has the next information:

- order_id: unique order identification.
- delivery_date: date when the order has to be complete. The day that is introduced is also counted as an available day to complete the product.
- quantity: quantity of product to be made.
- product_id: product identification.

An entry would have the next structure:

```
"orders":
[
 {
  "order_id":1,
  "delivery_date":"2020-01-12",
  "quantity":800,
  "product_id":1
 },
```

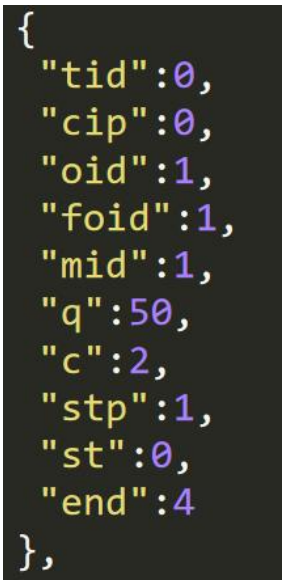
To have a complete example of an input file two files *test.json* and *test2.json* are provided.

Output File: (*Solution.json*:)

This file is the best solution found by the scheduler sorted by machine and time. Moreover, it also includes the worker assignation for the different shifts and machines if the option is enabled. The first entry is the 'task_sequence' and shows the sorted lists of tasks. Each element has:

- tid: task identification.
- cip: 0: regular task. 1: cip task.
- oid: which order satisfies.
- mid: At which machine is performed.
- q: quantity.
- c: task cost.
- stp: which step is.
- St: starting time of the task.
- end: ending time of the task.

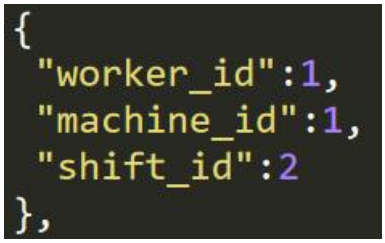
A task has the next architecture:



The second entry is 'workers_schedule' and shows the worker assignation. This entry has the next information:

- worker_id : worker identification.
- machine_id: machine identification designated.
- shift_id: shift identification designated.

An element of this entry is like:



IMPORANT NOTE: It is really important to maintain the definition of time consistent. All the parameters that involves time are integers. The interpretation of one unit of time is let to the user. One unit can be one hour, five minutes or whatever the user choose with the only restriction that all the parameter declaration have to be made with the same definition of time. Keep in mind that if more resolution of time is required the scheduler will take more time to solve the problem, so a smart choice is encouraged.

3.User manual.

In order to being able to use this tool is required to install **python 3.7** and the following libraries:
pandas 1.0.2, ortools 7.4.7247, matplotlib 3.1.3, numpy 1.18.1, json 0.9.4, flask 1.1.2.

Some of them should be installed with the installation of python. The version specified of each library is the version which the scheduler has been developed with, other versions are not tested.

3.1 Standalone application

If the program has to be run alone on a single computer, the file main_standalone.py has to be executed. Lets see in detail the file:

```
from scheduler import ProblemDef as PD

Pro = PD.Problem('ProvesWorkers1.json','json')
Pro.Scheduler()

Pro.Graphic_Sequence()

Pro.Return_json()
```

The first line imports all the functionalities of the scheduler. The second line creates the instance of the problem with the JSON file. The first argument of this initialization is the file path of the input file and the second argument is the working mode, in this case ‘json’ for the standalone mode.

With Scheduler(), it calls the scheduler to solve our problem instance. With Graphic_Sequence() we get the graphic representation of the solution and with Return_json() we create the output file. Note: when we call Return_json() the solution is manipulated and some information is eliminated to give a concise solution and the Graphic_sequence() method will not be valid. The order of the calls are important.

3.2 API application

First of all we need to execute the file Flask on the server inside the folder Scheduler. If the initialization is correct, we will see something similar to:

```
* Serving Flask app "Flask" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 808-907-042
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

That means that the scheduler as a service is ready.

Now we have to send the data (same format as the input file) to the direction http://127.0.0.1:5000/scheduler where 127.0.0.1:5000 is the IP direction where the scheduler is running. The call is through a POST method.

When the scheduler is done, it will send the solution with the same format as the output file explained above.

If we take a closer look to the Flask.py module to understand how it works:

```
import flask
from flask import request, jsonify
from scheduler import ProblemDef as PD

app = flask.Flask(__name__)
app.config["DEBUG"] = True
```

```
@app.route('/scheduler', methods=['POST'])
def scheduler():
        data = request.get_json()
        Pro = PD.Problem(data,'raw')
        Pro.Scheduler()

        return Pro.Return_json()


if __name__ == '__main__':
    app.run()
```

First lets analyze how the API is build:

- import flask: import the library Flask.
- app = flask.Flask (__ name__): Creates the object of the flask application that contains usefull methods.
- app.config ["DEBUG"] = True: starts the debugg mode, it shows it returns some info in the browser if the code have some error.
- @app.route('/', methods=['Post']): It assinges the URL to functions (routing).
- app.run (): Is the method that executes the server with its applications.

Inside the function scheduler, first we read the information through request.get_json(). This information has the same format as the input file. Then, the object which is the instance of the problem is initialized and solved in the same manner as the standalone application. Finally it returns the solution with the method Return_json().

4. General Structure of the code.

Inside the folder scheduler is all the necessary code for the scheduler. First of all we have the file ProblemDef.py where the definition of the class problem is defined. This class contains all the parameters and tables necessary along with the solver and pre- and post-processing methods. Inside this file it can be seen clearly the structure of the

scheduler. The main methods are the last three, the scheduler itself, the Graphic_Sequence() and the Return_Json() which are explained above.

The scheduler method is performed with the following files. They appear in order of execution.

- ProblemProcessing: inside this file is all the required pre-processing of the problem instance. It takes all the information given by the input file and transforms it to multiple pandas dataframes. It takes into account the machines availability, the tasks that have to be performed to a given order to fulfill it, etc...
- ConstructiveStep: Here the constructive step is performed. Here the scheduler chooses which formula is best if a multistage formula is detected.
- ModelSolver: Here the constraints are defined and the solver is called. Moreover it has the useful tool to define the subsequences needed (More explained on the thesis). This subsequence definition is the last step before calling the solver and it is a very delicate and vital step, that is why it is on this file, because it gives more perspective of when it is applied and its communication between other functions.
- PostProcessing: Here the cip task are added, and the second solver is called. It contains all the post-processing functions.
- GenerativeStep: Here the Reinforcement is implemented.
- WorkersProcessing: The workers scheduler is defined (More details on section 5)

Note that all the main functionalities (encoding, generative, solver) is defined at the end of the file. Also note that if the problem is unfeasible the dataframe T_single is set to an integer not a dataframe, this is a control method of the class to control if the problem is feasible. If the ‘workers’ option is disabled, the workers schedule is also an integer transforming the input file to not contain the worker_schedule.

For the Graphic_Sequence() the file Scheduleplot is used. This file contains how the solution is represented with matplotlib.

Return_Json() uses the file JsonTools. This files contains all the functionalities associated with reading and extracting the information from JSON files and convert it to a pandas dataframe. It also contains the code to write the output file.

For a more detailed explanation of the algorithm go to the thesis report attached.

6.Worker Scheduler

The worker schedule is not explained on the master thesis and that is why it is explained briefly here.

The worker scheduler modifies the list of machines. It deletes the least efficient machines according to the minimum worker available on all the possible shifts. This filtering is performed on the ProblemProcessing file code. This is done to ensure a feasible schedule. For the deletion of the machines the user preferences are taken into account, first the machines on a given multistage step are scores and this scores are normalized. When all the multistage steps machines are scored, the mean value of this score is taken for each machine in the case that a machine take part of different multistage steps. With this method we ensure the elimination of the least efficient machine taking into account the user preferences.

Once a feasible schedule is obtained the workers are assigned to available shifts and machines. The assignation is made with the policy of accumulated working time. It will be designated always the worker that have the least accumulated working time, letting the most scheduled workers out of the shift.