JSSP

Release 2.1.0

Jul 02, 2020

Contents

1 Problem Description **3**

2 Constraints **5**

3 Solution Formulation **7**

4 Input Data **9**
4.1 FJS Instances with Sequence Dependent Setup Times . . . 9
4.2 FJS Instances . . . 10

5 Getting Started **13**
5.1 How to Install . . . 13
5.2 How to Use . . . 13

6 Examples **17**
6.1 Tabu Search & Genetic Algorithm Example . . . 17
6.2 Gantt Chart Example . . . 18
6.3 Benchmark Example . . . 18

7 API Documentation **21**
7.1 JSSP package . . . 21

8 Welcome to JSSP's Documentation! **33**
8.1 Features . . . 33

9 Indices and Tables **35**

Python Module Index **37**

Index **39**

The specific job shop schedule problem JSSP was made to solve is classified as the **Partial Flexible Job Shop Scheduling Problem with Sequence Dependent Setup Times**. For more general information on job shop scheduling problems see https://en.wikipedia.org/wiki/Job_shop_scheduling.

CHAPTER 1

Problem Description

Given m machines with different run speeds and n jobs which have a varying number of tasks that need to be completed, schedule the tasks from all jobs on the machines such that the completion time (i.e. makespan) is minimized.

Additional Information

- Each task has a set amount of pieces that needs to be processed for it to be complete.
- The machines all have different run speeds.
- Certain tasks can be run in parallel on these machines.
- Each task has a sequence dependent setup time. Setup times are encoded in sequenceDependencyMatrix.csv.
- If a task is the first to run on a machine there will be no setup time.

CHAPTER 2

Constraints

- On a machine, there can be no overlap in start and end times for any job-task.
- Tasks within the same job cannot be started unless all other tasks within the same job with a sequence number less than the current task are complete.
- Each task can only be run on certain machines.

CHAPTER 3

Solution Formulation

A solution is formulated as a vector of operations where an operation is a vector consisting of job ID, task ID, sequence number, and machine number. The order of the operations (rows) determines the order in which the operations are scheduled. For example, below is a feasible solution to a problem instance with 3 jobs, and 2 machines. Each row represents an operation in the form [job id, task id, sequence number, machine number].

```
[[0, 0, 0, 0],
 [0, 1, 1, 1],
 [1, 0, 0, 1],
 [2, 0, 0, 0],
 [1, 1, 1, 0]]
```

Here is an example of an infeasible solution:

```
[[0, 1, 1, 0],
 [0, 0, 0, 0],
 [1, 0, 0, 1],
 [2, 0, 0, 0],
 [1, 1, 1, 0]]
```

The solution is infeasible because job 0, task 1 with sequence # = 1 is scheduled on machine 0 before job 0, task 0 with sequence # = 0.

CHAPTER 4

Input Data

The input data for partial flexible job shop schedule problem instances with sequence dependent setup times is encoded in three csv files. The input data for flexible job shop schedule problem instances is encoded in a `.fjs` file.

To see the full example files, view them on GitHub.

4.1 FJS Instances with Sequence Dependent Setup Times

4.1.1 jobTasks.csv

Contains a table of all job-tasks and their sequence numbers, usable machines, and pieces. **Job-tasks need to be in ascending order according to (job_id, task_id)**

Job	Task	Sequence	Usable_Machines	Pieces
0	0	0	[4 5]	19116
0	1	1	[0 1 2 4 5 6 7]	7897
0	2	2	[0 1 2]	23430
0	3	2	[2 7]	21321
0	4	3	[0 3 6]	15368
0	5	3	[3 4]	23025
0	6	4	[2 3 4 6 7]	20115
1	0	0	[3 6]	22455
1	1	1	[3 6]	19165
1	2	1	[2]	8762
1	3	2	[5 7]	18824
1	4	2	[0 2]	9305
2	0	0	[0 3 4]	19368
2	1	1	[1 4 5 6]	20875
2	2	1	[1 2 4 6]	12997
2	3	1	[2 3 4 6]	6134

4.1.2 machineRunSpeed.csv

Contains a list of all machine IDs and run speeds.

Machine	RunSpeed
0	123
1	58
2	76
3	83
4	111
5	100
6	98
7	151

4.1.3 sequenceDependencyMatrix.csv

Contains a matrix of job-task setup times. **Job-tasks need to be in ascending order according to (job_id, task_id)**

.	0_0	0_1	0_2	0_3	0_4	0_5	0_6	1_0	1_1	1_2	1_3	1_4	2_0	2_1	2_2
0_0	-1	-1	-1	-1	-1	-1	-1	5	2	3	1	1	8	7	3
0_1	1	-1	-1	-1	-1	-1	-1	8	2	1	8	7	8	1	5
0_2	-1	7	-1	4	-1	-1	-1	8	8	6	3	2	8	6	1
0_3	-1	2	6	-1	-1	-1	-1	3	2	3	7	8	5	7	2
0_4	-1	-1	2	3	-1	2	-1	5	8	9	5	5	7	6	2
0_5	-1	-1	5	4	2	-1	-1	2	3	9	2	8	9	3	7
0_6	-1	-1	-1	-1	9	9	-1	5	1	5	3	8	5	5	6
1_0	7	7	8	1	6	5	8	-1	-1	-1	-1	-1	5	7	7
1_1	4	4	5	2	3	5	9	2	-1	4	-1	-1	1	8	9
1_2	7	4	8	9	5	9	3	7	7	-1	-1	-1	6	8	9
1_3	3	6	2	7	7	1	4	-1	8	1	-1	7	9	3	9
1_4	8	9	3	5	7	1	8	-1	9	6	5	-1	4	6	9
2_0	3	1	7	9	6	8	4	6	1	4	1	9	-1	-1	-1
2_1	3	2	4	3	6	8	1	7	7	2	1	7	6	-1	8
2_2	5	5	3	9	3	2	6	8	6	5	7	6	4	9	-1

The rows of sequenceDependencyMatrix.csv represent the current job-task in the form *j_t* where *j* is the job ID and *t* is the task ID. The columns represent the previous job-task. A matrix value at index *(j_t, x_y)* represents the setup time to schedule job-task *j_t* after job-task *x_y* on a machine. Matrix values with the value -1 represent cases where the current job-task (row) cannot be scheduled after the previous job-task (column).

4.2 FJS Instances

In the first line there are (at least) 2 numbers:

1. the number of jobs
2. the number of machines
3. the average number of machines per operation (optional)

Every row represents one job:

- the first number is the number of operations of that job
- the second number (let's say k>=1) is the number of machines that can process the first operation; then according to k, there are k pairs of numbers (machine, processing time) that specify which are the machines and the processing times

Example: Fisher and Thompson 6x6 instance, alternate name (mt06)

```
6   6   1
6   1   3   1   1   1   3   1   2   6   1   4   7   1   6   3   1   5   6
6   1   2   8   1   3   5   1   5   10  1   6   10  1   1   10  1   4   4
6   1   3   5   1   4   4   1   6   8   1   1   9   1   2   1   1   5   7
6   1   2   5   1   1   5   1   3   5   1   4   3   1   5   8   1   6   9
6   1   3   9   1   2   3   1   5   5   1   6   4   1   1   3   1   4   1
6   1   2   3   1   4   3   1   6   9   1   1   10  1   5   4   1   3   1
```

first row: 6 jobs, 6 machines, and 1 machine per operation second row: job 1 has 6 operations; the first operation can be processed by 1 machine, that is machine 3 with processing time 1.

CHAPTER 5

Getting Started

5.1 How to Install

1. Download the latest stable release of JSSP from GitHub
2. Run `easy_install JSSP-<release>.<os>.egg`

If you downloaded the source code, `cd` to the directory where `setup.py` is and run `pip install .`

If you get an error about `python.h` not being found try installing python3-dev.

5.2 How to Use

After installation, JSSP can imported as a normal python module.

Important Note Job-tasks in jobTasks.csv and sequenceDependencyMatrix.csv need to be in ascending order according to (job_id, task_id). (see the csv files on GitHub for a reference)

For more information on the input data see the Input Data section.

5.2.1 Read Data

To read in a partial flexible job shop schedule problem instance with sequence dependent setup times (i.e. `.csv` files), run the following:

```
from JSSP.data import CSVData

data = CSVData('sequenceDependencyMatrix.csv',
               'machineRunSpeed.csv',
               'jobTasks.csv')
```

To read in a flexible job shop schedule problem instance (i.e. `.fjs` file), run the following:

```
from JSSP.data import FJSData

data = FJSData('Brandimarte_Mk10.fjs')
```

5.2.2 Optimization

To run an optimization algorithm first create a `Solver` instance:

```
from JSSP.solver import Solver

solver = Solver(data)
```

Next, run the optimization algorithm:

```
# runs 4 parallel tabu search processes for 500 iterations each
solution = solver.tabu_search_iter(iterations=500,
                                   num_processes=4,
                                   tabu_list_size=15,
                                   neighborhood_size=250,
                                   neighborhood_wait=0.1,
                                   probability_change_machine=0.8,
                                   reset_threshold=100,
                                   )
```

Note: See the Solver module for more optimization functions and parameter options.

5.2.3 Output

Now that you have a Solution object run the following to produce a production schedule (excel file):

```
import datetime

solution.create_schedule_xlsx_file(output_path='./Schedule.xlsx',
                                   start_date=datetime.date.today(),
                                   start_time=datetime.time(8, 0),
                                   end_time=datetime.time(20, 0))
```

Below is only a portion of the produced `Schedule.xlsx`. To view the full schedule download it.

Machine 0			Machine 1		
Makespan =	15 days, 19:24:51		Makespan =	9 days, 19:05:14	
Job_Task	Start	End	Job_Task	Start	End
48_0 setup	2020-02-29 08:00:00	2020-02-29 08:00:00	28_1 setup	2020-02-29 08:00:00	2020-02-29 08:00:00
48_0 run	2020-02-29 08:00:00	2020-02-29 10:20:36	28_1 run	2020-02-29 08:00:00	2020-02-29 10:40:15
4_2 setup	2020-02-29 10:20:36	2020-02-29 10:21:36	19_0 setup	2020-02-29 10:40:15	2020-02-29 10:45:15
4_2 run	2020-02-29 10:21:36	2020-02-29 11:23:04	19_0 run	2020-02-29 10:45:15	2020-02-29 17:33:45
11_2 setup	2020-02-29 11:23:04	2020-02-29 11:24:04	40_0 setup	2020-03-01 17:33:45	2020-03-01 17:33:45
11_2 run	2020-02-29 11:24:04	2020-02-29 13:18:12	40_0 run	2020-03-01 17:33:45	2020-03-01 23:11:31
31_0 setup	2020-02-29 13:18:12	2020-02-29 13:21:12	30_0 setup	2020-03-03 23:11:31	2020-03-03 23:11:31
31_0 run	2020-02-29 13:21:12	2020-02-29 16:33:53	30_0 run	2020-03-03 23:11:31	2020-03-04 01:17:44
8_0 setup	2020-02-29 16:33:53	2020-02-29 16:34:53	4_0 setup	2020-03-04 01:17:44	2020-03-04 01:18:44
8_0 run	2020-02-29 16:34:53	2020-02-29 17:54:21	4_0 run	2020-03-04 01:18:44	2020-03-04 02:47:17

Continued on next page

Table 1 – continued from previous page

10_0 setup	2020-02-29 17:54:21	2020-02-29 17:56:21	10_1 setup	2020-03-04 02:47:17	2020-03-04 02:48:17
10_0 run	2020-02-29 17:56:21	2020-02-29 19:13:43	10_1 run	2020-03-04 02:48:17	2020-03-04 05:36:09
49_1 setup	2020-03-01 19:13:43	2020-03-01 19:13:43	45_1 setup	2020-03-04 05:36:09	2020-03-04 05:37:09
49_1 run	2020-03-01 19:13:43	2020-03-01 22:25:26	45_1 run	2020-03-04 05:37:09	2020-03-04 07:08:35
32_1 setup	2020-03-03 22:25:26	2020-03-03 22:25:26	37_3 setup	2020-03-04 07:08:35	2020-03-04 07:11:35
32_1 run	2020-03-03 22:25:26	2020-03-03 23:45:17	37_3 run	2020-03-04 07:11:35	2020-03-04 08:46:16
25_0 setup	2020-03-05 23:45:17	2020-03-05 23:45:17	29_0 setup	2020-03-04 08:46:16	2020-03-04 08:47:16
25_0 run	2020-03-05 23:45:17	2020-03-06 02:39:25	29_0 run	2020-03-04 08:47:16	2020-03-04 10:29:46
13_0 setup	2020-03-06 02:39:25	2020-03-06 02:40:25	44_1 setup	2020-03-04 10:29:46	2020-03-04 10:32:46
13_0 run	2020-03-06 02:40:25	2020-03-06 05:01:55	44_1 run	2020-03-04 10:32:46	2020-03-04 13:39:18
28_3 setup	2020-03-06 05:01:55	2020-03-06 05:05:55	6_0 setup	2020-03-04 13:39:18	2020-03-04 13:43:18
28_3 run	2020-03-06 05:05:55	2020-03-06 07:01:30	6_0 run	2020-03-04 13:43:18	2020-03-04 16:42:47
5_3 setup	2020-03-06 07:01:30	2020-03-06 07:02:30	28_4 setup	2020-03-05 16:42:47	2020-03-05 16:42:47
5_3 run	2020-03-06 07:02:30	2020-03-06 09:00:23	28_4 run	2020-03-05 16:42:47	2020-03-05 20:44:03
9_1 setup	2020-03-06 09:00:23	2020-03-06 09:01:23	12_2 setup	2020-03-07 20:44:03	2020-03-07 20:44:03
9_1 run	2020-03-06 09:01:23	2020-03-06 10:35:41	12_2 run	2020-03-07 20:44:03	2020-03-08 00:39:12
22_1 setup	2020-03-06 10:35:41	2020-03-06 10:36:41	48_4 setup	2020-03-08 00:39:12	2020-03-08 00:40:12
22_1 run	2020-03-06 10:36:41	2020-03-06 12:06:51	48_4 run	2020-03-08 00:40:12	2020-03-08 03:07:43

CHAPTER 6

Examples

This document contains examples for how to use JSSP. For more in depth examples see the jupyter notebook files in the examples folder on GitHub.

6.1 Tabu Search & Genetic Algorithm Example

The example below demonstrates how to utilize both parallel tabu search and the genetic algorithm to solve an instance of a job shop schedule problem. In the example, after initializing the data object from the three csv files, 4 tabu search processes are ran in parallel with each returning 5 solutions. Next the resulting solutions from parallel tabu search are added to the initial population for the genetic algorithm. The `solver.genetic_algorithm_time` function adds randomly generated solutions to the initial population until it has `population_size` solutions. Lastly, an excel file of the production schedule (i.e. solution) is created in the `./example_output` directory.

```
from JSSP.data import CSVData
from JSSP import Solver
from JSSP.genetic_algorithm import GASelectionEnum

# initialize data
data = CSVData('data/given_data/sequenceDependencyMatrix.csv',
               'data/given_data/machineRunSpeed.csv',
               'data/given_data/jobTasks.csv')

# create solver
solver = Solver(data)

# run tabu seach
solver.tabu_search_time(runtime=30, # seconds
                        num_processes=4,
                        num_solutions_per_process=5,
                        tabu_list_size=15,
                        neighborhood_size=250,
                        neighborhood_wait=0.1,
                        probability_change_machine=0.8,
```

(continues on next page)

(continued from previous page)

```
                        reset_threshold=80
                        )

# add all tabu search solutions to population
population = []
for ts_agent in solver.ts_agent_list:
    population += ts_agent.all_solutions

# run genetic algorithm
solution = solver.genetic_algorithm_time(runtime=30, # seconds
                                         population_size=100,
                                         selection_method_enum=GASelectionEnum.
↪FITNESS_PROPORTIONATE,
                                         mutation_probability=0.1
                                         )

# create an excel file of the schedule
solution.create_schedule_xlsx_file('./Schedule.xlsx')
```

Output

Schedule.xlsx

Alternatively you can run either the genetic algorithm or parallel tabu search for a certain number of iterations instead of time - just use `solver.tabu_search_iter()` and/or `solver.genetic_algorithm_iter()`.

6.2 Gantt Chart Example

The example below demonstrates how to create a gantt chart given a Solution.

```
# create a gantt chart html file
solution.create_gantt_chart_html_file('./gantt_chart.html', continuous=True)

# alternatively you can plot a gantt chart in an ipython notebook
solution.iplot_gantt_chart(continuous=True)
```

Output

6.3 Benchmark Example

The example below demonstrates how to run a benchmark (i.e. create plots & statistical information for run) for both parallel tabu search and the genetic algorithm.

```
from JSSP.data import FJSData
from JSSP.genetic_algorithm import GASelectionEnum
from JSSP.solution import SolutionFactory

# initialize fjs data
data = FJSData('data/fjs_data/Brandimarte/Brandimarte_Mk10.fjs')

# ts parameters
ts_iterations = 200
num_solutions_per_process = 20
```

(continues on next page)

(continued from previous page)

```
num_processes = 5
tabu_list_size = 15
neighborhood_size = 300
neighborhood_wait = 0.15
probability_change_machine = 0.8

# ga parameters
ga_iterations = 200
population_size = 400
selection_method = GASelectionEnum.FITNESS_PROPORTIONATE
selection_size = 10
mutation_probability = 0.2

# create solver
solver = Solver(data)

# run tabu search
solver.tabu_search_iter(ts_iterations,
                        num_solutions_per_process=num_solutions_per_process,
                        num_processes=num_processes,
                        tabu_list_size=tabu_list_size,
                        neighborhood_size=neighborhood_size,
                        neighborhood_wait=neighborhood_wait,
                        probability_change_machine=probability_change_machine,
                        verbose=True,
                        benchmark=True
                        )

# add all ts solutions to population
population = []
for ts_agent in solver.ts_agent_list:
    population += ts_agent.all_solutions

solution_factory = SolutionFactory(data)

# add 25% spt solutions to population
population += solution_factory.get_n_shortest_process_time_first_solution(int(.25 *␣
↪population_size))

# add 25% lpt solutions to population
population += solution_factory.get_n_longest_process_time_first_solution(int(.25 *␣
↪population_size))

# add 25% random solutions to population
population += solution_factory.get_n_solutions(int(.25 * population_size))

# run genetic algorithm
solver.genetic_algorithm_iter(ga_iterations,
                              population=population,
                              population_size=population_size,
                              selection_method_enum=selection_method,
                              mutation_probability=mutation_probability,
                              selection_size=selection_size,
                              verbose=True,
                              benchmark=True
                              )
```

(continues on next page)

(continued from previous page)

```
# output benchmark results
solver.output_benchmark_results('./example_benchmark', name='Example Benchmark')

# alternatively you can output the results in an ipython notebook
solver.iplot_benchmark_results()
```

Output

```
Running benchmark of TS
Parameters:
stopping_condition = 200 iterations
time_condition = False
num_solutions_per_process = 20
num_processes = 5
tabu_list_size = 15
neighborhood_size = 300
neighborhood_wait = 0.15
probability_change_machine = 0.8
reset_threshold = 100

Initial Solution's makespans:
[613, 730, 684, 671, 633]

child TS process started. pid = 6453
child TS process started. pid = 6454
child TS process started. pid = 6462
child TS process started. pid = 6463
child TS process started. pid = 6464
child TS process finished. pid = 6453
child TS process finished. pid = 6454
child TS process finished. pid = 6462
child TS process finished. pid = 6463
child TS process finished. pid = 6464
Running benchmark of GA
Parameters:
stopping_condition = 200 iterations
time_condition = False
population_size = 400
selection_method = _fitness_proportionate_selection
mutation_probability = 0.2
```

To view the benchmark results see example_benchmark.

CHAPTER 7

API Documentation

Information on specific functions, classes, and methods.

7.1 JSSP package

7.1.1 Subpackages

JSSP.genetic_algorithm package

Submodules

JSSP.genetic_algorithm.ga module

class JSSP.genetic_algorithm.ga.**GASelectionEnum**

Bases: enum.Enum

Enumeration class containing three selection methods for selecting parent solutions for the genetic algorithm.

Selection Methods:

1. GASelectionEnum.TOURNAMENT - Tournament style selection
2. GASelectionEnum. FITNESS_PROPORTIONATE - Fitness proportionate selection (also called roulette wheel selection)
3. GASelectionEnum.RANDOM - Random selection

FITNESS_PROPORTIONATE()

Fitness proportionate selection for the genetic algorithm (also called roulette wheel selection).

This function first normalizes the fitness values (makespan) of the solutions in the population, then randomly removes a solution from the population and returns it.

See https://en.wikipedia.org/wiki/Fitness_proportionate_selection for more info.

Parameters **args** – list where args[0] is the population

Return type *Solution*

Returns a Solution from the population

RANDOM()

Random selection for the genetic algorithm.

This function randomly removes a solution from the population and returns it.

Parameters **args** – list where args[0] is the population

Return type *Solution*

Returns a solution from the population

TOURNAMENT()

Tournament style selection for the genetic algorithm.

This function selects args[1] (i.e. selection_size) solutions from the population, then removes the best solution out of the selection from the population and returns it.

See https://en.wikipedia.org/wiki/Tournament_selection for more info.

Parameters **args** – list where args[0] is the population of solutions and args[1] is the selection size

Return type *Solution*

Returns a Solution from the population

class JSSP.genetic_algorithm.ga.**GeneticAlgorithmAgent**(*stopping_condition*, *population*, *time_condition=False*, *selection_method_enum=<function _tournament_selection>*, *mutation_probability=0.8*, *selection_size=2*, *benchmark=False*)

Bases: `object`

Genetic algorithm optimization agent.

Parameters

- **stopping_condition** (*float*) – either the duration to run GA in seconds or the number of generations to iterate though
- **population** ([Solution]) – list of Solutions to start the GA from, must not be empty
- **time_condition** (*bool*) – if true GA is ran for stopping_condition number of seconds else it is ran for stopping_condition generations
- **selection_method_enum** (GASelectionEnum) – selection method to use for selecting parents from the population. (see GASelectionEnum for selection methods)
- **mutation_probability** (*float*) – probability of mutating a child solution (i.e change a random operation's machine) in range [0, 1]
- **selection_size** (*int*) – size of the selection group. (applicable only for tournament style selection)
- **benchmark** (*bool*) – if true benchmark data is gathered

start()
Starts the genetic algorithm for this GeneticAlgorithmAgent.

Return type *Solution*

Returns best Solution found

JSSP.solution package

Submodules

JSSP.solution.factory module

class JSSP.solution.factory.**SolutionFactory**(*data*)
Bases: object

Factory class for generating Solution instances.

Parameters **data** (Data) – JSSP instance data

get_longest_process_time_first_solution()
Gets a random Solution instance that is generated using longest processing time first criteria.

Return type *Solution*

Returns randomly generated Solution instance

get_n_longest_process_time_first_solution(*n*)
Gets n random Solution instances that are generated using longest processing time first criteria.

Parameters **n** (*int*) – number of Solutions to get

Return type [*Solution*]

Returns n randomly generated Solution instances

get_n_shortest_process_time_first_solution(*n*)
Gets n random Solution instances that are generated using shortest processing time first criteria.

Parameters **n** (*int*) – number of Solutions to get

Return type [*Solution*]

Returns n randomly generated Solution instances

get_n_solutions(*n*)
Gets n random Solution instances.

Parameters **n** (*int*) – number of Solutions to get

Return type [*Solution*]

Returns n randomly generated Solution instances

get_shortest_process_time_first_solution()
Gets a random Solution instance that is generated using shortest processing time first criteria.

Return type *Solution*

Returns randomly generated Solution instance

get_solution()
Gets a random Solution instance.

Return type *Solution*

Returns randomly generated Solution instance

class `JSSP.solution.factory.`**`TaskWrapper`**(*data*, *val*)
Bases: `object`

JSSP.solution.solution module

class `JSSP.solution.solution.`**`Operation`**(*job_id*, *task_id*, *machine*, *wait*, *setup*, *runtime*, *start_time*)
Bases: `object`

class `JSSP.solution.solution.`**`Solution`**(*data*, *operation_2d_array*)
Bases: `object`

Solution class which is composed of a 2d nparray of operations where an operation is a 1d nparray in the form [job_id, task_id, sequence, machine], a 1d nparray memory view of machine makespan times, and the makespan time.

Parameters

- **data** (`Data`) – JSSP instance data
- **operation_2d_array** (`nparray`) – 2d nparray of operations

create_gantt_chart_html_file(*output_path*, *title='Gantt Chart'*, *start_date=datetime.date(2020, 7, 2)*, *start_time=datetime.time(8, 0)*, *end_time=datetime.time(20, 0)*, *auto_open=False*, *continuous=False*)
Creates a gantt chart html file of the solution.

Parameters

- **output_path** (`Path | str`) – path to the gantt chart html file to create
- **title** (`str`) – name of the gantt chart
- **start_date** (`datetime.date`) – date to start the schedule from
- **start_time** (`datetime.time`) – start time of the work day
- **end_time** (`datetime.time`) – end time of the work day
- **auto_open** (`bool`) – if true the gantt chart html file is automatically opened in a browser
- **continuous** (`bool`) – if true a continuous schedule is created. (i.e. start_time and end_time are not used)

Returns None

create_schedule_xlsx_file(*output_path*, *start_date=datetime.date(2020, 7, 2)*, *start_time=datetime.time(8, 0)*, *end_time=datetime.time(20, 0)*, *continuous=False*)
Creates an excel file that contains the schedule for each machine of this Solution.

Parameters

- **output_path** (`Path | str`) – path to the excel file to create
- **start_time** (`datetime.time`) – start date of the schedule
- **start_time** – start time of the work day
- **end_time** (`datetime.time`) – end time of the work day

- **continuous** (*bool*) – if true a continuous schedule is created. (i.e. start_time and end_time are not used)

Returns None

get_operation_list_for_machine(*start_date=datetime.date(2020, 7, 2), start_time=datetime.time(8, 0), end_time=datetime.time(20, 0), continuous=False, machines=None*)

Gets a list of Operations for a machine or set of machines.

Parameters

- **start_date** (*datetime.date*) – date to start the schedule from
- **start_time** (*datetime.time*) – start time of the work day
- **end_time** (*datetime.time*) – end time of the work day
- **continuous** (*bool*) – if true a continuous schedule is created. (i.e. start_time and end_time are not used)
- **machines** (*[int]*) – list of machine ids, or None

Return type [*Operation*]

Returns list of Operations

iplot_gantt_chart(*title='Gantt Chart', start_date=datetime.date(2020, 7, 2), start_time=datetime.time(8, 0), end_time=datetime.time(20, 0), continuous=False*)

Plots a gantt chart of this Solution in an ipython notebook.

Parameters

- **title** (*str*) – name of the gantt chart
- **start_date** (*datetime.date*) – date to start the schedule from
- **start_time** (*datetime.time*) – start time of the work day
- **end_time** (*datetime.time*) – end time of the work day
- **continuous** (*bool*) – if true a continuous schedule is created. (i.e. start_time and end_time are not used)

Returns None

JSSP.tabu_search package

Submodules

JSSP.tabu_search.ts module

class JSSP.tabu_search.ts.**TabuSearchAgent**(*stopping_condition, time_condition, initial_solution, num_solutions_to_find=1, tabu_list_size=50, neighborhood_size=300, neighborhood_wait=0.1, probability_change_machine=0.8, reset_threshold=100, benchmark=False*)

Bases: object

Tabu search optimization agent.

Parameters

- **stopping_condition** (*float*) – either the duration in seconds or the number of iterations to search
- **time_condition** (*bool*) – if true TS is ran for stopping_condition number of seconds else it is ran for stopping_condition number of iterations
- **initial_solution** (Solution) – initial solution to start the tabu search from
- **num_solutions_to_find** (*int*) – number of best solutions to find
- **tabu_list_size** (*int*) – size of the Tabu list
- **neighborhood_size** (*int*) – size of neighborhoods to generate during each iteration
- **neighborhood_wait** (*float*) – maximum time (in seconds) to wait while generating a neighborhood
- **probability_change_machine** (*float*) – probability of changing a chosen operations machine, must be in range [0, 1]
- **reset_threshold** (*int*) – number of iterations to potentially force a worse move after if the best solution is not improved
- **benchmark** (*bool*) – if true benchmark data is gathered

start (*multi_process_queue=None*)

Starts the search for this TabuSearchAgent.

If the multi_process_queue parameter is not None, this function attempts to push this TabuSearchAgent to the multi processing queue.

Parameters **multi_process_queue** (*multiprocessing.Queue*) – queue to put this TabuSearchAgent into

Return type *Solution*

Returns best Solution found

7.1.2 Submodules

JSSP.benchmark_plotter module

JSSP.benchmark_plotter.**iplot_benchmark_results** (*ts_agent_list=None*, *ga_agent=None*)

Plots the benchmark results in an ipython notebook.

Parameters

- **ts_agent_list** ([TabuSearchAgent]) – list of TabuSearchAgent instances to plot the benchmark results for
- **ga_agent** (GeneticAlgorithmAgent) – GeneticAlgorithmAgent to plot the results for

Returns None

JSSP.benchmark_plotter.**output_benchmark_results** (*output_dir*, *ts_agent_list=None*, *ga_agent=None*, *title=None*, *auto_open=True*)

Outputs html files containing benchmark results in the output directory specified.

Parameters

- **ts_agent_list** (*[TabuSearchAgent]*) – list of TabuSearchAgent instances to output the benchmark results for
- **ga_agent** (*GeneticAlgorithmAgent*) – GeneticAlgorithmAgent instance to output the benchmark results for
- **output_dir** (*Path | str*) – path to the output directory to place the html files into
- **title** (*str*) – name of the benchmark run, default to current datetime
- **auto_open** (*bool*) – if true the benchmark output is automatically opened in a browser

Returns None

JSSP.data module

class JSSP.data.**Data**

Bases: abc.ABC

Base class for JSSP instance data.

static convert_fjs_to_csv (*fjs_file*, *output_dir*)

Converts a fjs file into three csv files, jobTasks.csv, machineRunSpeed.csv, and sequenceDependencyMatrix.csv, then it puts them in the output directory.

Parameters

- **fjs_file** (*Path | str*) – path to the fjs file containing a flexible job shop schedule problem instance
- **output_dir** (*Path | str*) – path to the directory to place the csv files into

Returns None

get_job (*job_id*)

Gets the Job with job id = job_id.

Parameters **job_id** (*int*) – id of the Job to get

Return type *Job*

Returns Job with id = job_id

get_runtime (*job_id*, *task_id*, *machine*)

Gets the run time for running (job_id, task_id) on machine.

Parameters

- **job_id** (*int*) – job id
- **task_id** (*int*) – task id
- **machine** (*int*) – id of machine

Return type float

Returns run time

get_setup_time (*job1_id*, *job1_task_id*, *job2_id*, *job2_task_id*)

Gets the setup time for scheduling (job2_id, job2_task_id) after (job1_id, job1_task_id).

Parameters

- **job1_id** (*int*) – job id of job 1
- **job1_task_id** (*int*) – task id of job 1

- **job2_id** (*int*) – job id of job 2
- **job2_task_id** (*int*) – task id of job 2

Return type int

Returns setup time in minutes

job_task_index_matrix = None
2d nparray of (job, task): index mapping

jobs = None
list of all Job instances

machine_speeds = None
1d nparray of machine speeds

sequence_dependency_matrix = None
2d nparray of sequence dependency matrix

task_processing_times_matrix = None
2d nparray of task processing times on machines

usable_machines_matrix = None
2d nparray of usable machines

class JSSP.data.**FJSData**(*input_file*)
Bases: *JSSP.data.Data*

JSSP instance data class for .fjs (Flexible Job Shop) data.

Parameters **input_file** (*Path | str*) – path to the fjs file to read the data from

Returns None

class JSSP.data.**Job**(*job_id*)
Bases: object

Job ADT.

Parameters **job_id** (*int*) – job ID of this Job

get_job_id()

get_max_sequence()

get_number_of_tasks()

get_task(*task_id*)

get_tasks()

set_max_sequence(*max_sequence*)

class JSSP.data.**SpreadsheetData**(*seq_dep_matrix*, *machine_speeds*, *job_tasks*)
Bases: *JSSP.data.Data*

JSSP instance data class for spreadsheet data.

Parameters

- **seq_dep_matrix** (*Path | str | Dataframe*) – path to the csv or xlsx file or a Dataframe containing the sequence dependency setup times
- **machine_speeds** (*Path | str | Dataframe*) – path to the csv or xlsx file or a Dataframe containing all of the machine speeds

- **job_tasks** (*Path | str | Dataframe*) – path to the csv or xlsx file or a Dataframe containing all of the job-tasks

Returns None

class JSSP.data.**Task**(*job_id*, *task_id*, *sequence*, *usable_machines*, *pieces*)

Bases: object

Task ADT.

Parameters

- **job_id** (*int*) – job ID of this Task
- **task_id** (*int*) – task ID of this Task
- **sequence** (*int*) – sequence number of this Task
- **usable_machines** (*1d nparray*) – usable machines that this Task can be processed on
- **pieces** (*int*) – number of pieces this Task has

get_job_id()

get_pieces()

get_sequence()

get_task_id()

get_usable_machines()

JSSP.exception module

exception JSSP.exception.**IncompleteSolutionException**

Bases: Exception

exception JSSP.exception.**InfeasibleSolutionException**

Bases: Exception

JSSP.solver module

class JSSP.solver.**Solver**(*data*)

Bases: object

The main solver class which calls tabu search and/or the genetic algorithm.

Parameters **data** (Data) – JSSP instance data

genetic_algorithm_iter(*iterations*, *population=None*, *population_size=200*, *selection_method_enum=<function _tournament_selection>*, *mutation_probability=0.8*, *selection_size=10*, *benchmark=False*, *verbose=False*)

Performs the genetic algorithm for a certain number of generations.

First this function generates a random initial population if the population parameter is None, then it runs GA with the parameters specified and updates self.solution.

Parameters

- **iterations** (*int*) – number of generations to go through during the GA
- **population** ([Solution]) – list of Solutions to start the GA from

- **population_size** (*int*) – size of the initial population
- **selection_method_enum** (GASelectionEnum) – selection method to use for selecting parents from the population. Options are GASelectionEnum.TOURNAMENT, GASelectionEnum.FITNESS_PROPORTIONATE, GASelectionEnum.RANDOM
- **mutation_probability** (*float*) – probability of mutating a chromosome (i.e change an operation's machine), must be in range [0, 1]
- **selection_size** (*int*) – size of the selection group for tournament style selection
- **benchmark** (*bool*) – if true benchmark data is gathered (i.e. # of iterations, makespans, min makespan iteration)
- **verbose** (*bool*) – if true runs in verbose mode

Return type *Solution*

Returns best solution found

genetic_algorithm_time(*runtime*, *population=None*, *population_size=200*, *selection_method_enum=<function _tournament_selection>*, *mutation_probability=0.8*, *selection_size=10*, *benchmark=False*, *verbose=False*, *progress_bar=False*)

Performs the genetic algorithm for a certain number of seconds.

First this function generates a random initial population if the population parameter is None, then it runs GA with the parameters specified and updates self.solution.

Parameters

- **runtime** (*float | datetime.timedelta*) – either the number of seconds or timedelta to run the GA for
- **population** (*[Solution]*) – list of Solutions to start the GA from
- **population_size** (*int*) – size of the initial population
- **selection_method_enum** (GASelectionEnum) – selection method to use for selecting parents from the population. Options are GASelectionEnum.TOURNAMENT, GASelectionEnum.FITNESS_PROPORTIONATE, GASelectionEnum.RANDOM
- **mutation_probability** (*float*) – probability of mutating a chromosome (i.e change an operation's machine), must be in range [0, 1]
- **selection_size** (*int*) – size of the selection group for tournament style selection
- **benchmark** (*bool*) – if true benchmark data is gathered (i.e. # of iterations, makespans, min makespan iteration)
- **verbose** (*bool*) – if true runs in verbose mode
- **progress_bar** (*bool*) – if true a progress bar is spawned

Return type *Solution*

Returns best solution found

iplot_benchmark_results()

Plots the benchmark results in an ipython notebook.

Returns None

:raise UserWarning if one of the optimization functions was not ran in benchmark mode

output_benchmark_results(*output_dir*, *title=None*, *auto_open=True*)

Outputs html files containing benchmark results in the output directory specified.

Parameters

- **output_dir** (*Path | str*) – path to the output directory to place the results into
- **title** (*str*) – title of the benchmark run
- **auto_open** (*bool*) – if true index.html is automatically opened in a browser

Returns None

:raise UserWarning if one of the optimization functions was not ran in benchmark mode

tabu_search_iter(*iterations*, *num_solutions_per_process=1*, *num_processes=4*, *tabu_list_size=50*, *neighborhood_size=300*, *neighborhood_wait=0.1*, *probability_change_machine=0.8*, *reset_threshold=100*, *initial_solutions=None*, *benchmark=False*, *verbose=False*)

Performs parallel tabu search for a certain number of iterations.

First the function generates random initial solutions if the initial_solutions parameter is None, then it forks a number of child processes to run tabu search.

The parent process waits for the child processes to finish, then collects their results and updates self.solution.

Parameters

- **iterations** (*int*) – number of iterations for each tabu search to go through
- **num_solutions_per_process** (*int*) – number of solutions that one tabu search process should gather
- **num_processes** (*int*) – number of processes to run tabu search in parallel
- **tabu_list_size** (*int*) – size of the tabu list
- **neighborhood_size** (*int*) – size of neighborhoods to generate during tabu search
- **neighborhood_wait** (*float*) – maximum time to wait while generating a neighborhood in seconds
- **probability_change_machine** (*float*) – probability of changing a chosen operations machine, must be in range [0, 1]
- **reset_threshold** (*int*) – number of iterations to potentially force a worse move after if the best solution is not improved
- **initial_solutions** (*[Solution]*) – initial solutions to start the tabu searches from
- **benchmark** (*bool*) – if true benchmark data is gathered (e.g. # of iterations, makespans, etc.)
- **verbose** (*bool*) – if true runs in verbose mode

Return type *Solution*

Returns best solution found

tabu_search_time(*runtime*, *num_solutions_per_process=1*, *num_processes=4*, *tabu_list_size=50*, *neighborhood_size=300*, *neighborhood_wait=0.1*, *probability_change_machine=0.8*, *reset_threshold=100*, *initial_solutions=None*, *benchmark=False*, *verbose=False*, *progress_bar=False*)

Performs parallel tabu search for a certain number of seconds.

First the function generates random initial solutions if the initial_solutions parameter is None, then it forks/spawns num_processes of child processes to run tabu search in parallel.

The parent process waits for the child processes to finish, then collects their results and updates self.solution.

Parameters

- **runtime** (*float | datetime.timedelta*) – either the number of seconds or timedelta that tabu search should run for
- **num_solutions_per_process** (*int*) – number of solutions that one tabu search process should gather
- **num_processes** (*int*) – number of processes to run tabu search in parallel
- **tabu_list_size** (*int*) – size of the tabu list
- **neighborhood_size** (*int*) – size of neighborhoods to generate during tabu search
- **neighborhood_wait** (*float*) – maximum time to wait while generating a neighborhood in seconds
- **probability_change_machine** (*float*) – probability of changing a chosen operations machine, must be in range [0, 1]
- **reset_threshold** (*int*) – number of iterations to potentially force a worse move after if the best solution is not improved
- **initial_solutions** (*[Solution]*) – initial solutions to start the tabu searches from
- **benchmark** (*bool*) – if true benchmark data is gathered (e.g. # of iterations, makespans, etc.)
- **verbose** (*bool*) – if true runs in verbose mode
- **progress_bar** (*bool*) – if true a progress bar is spawned

Return type *Solution*

Returns best solution found

CHAPTER 8

Welcome to JSSP's Documentation!

JSSP is an optimization package for the Job Shop Schedule Problem. For more information on the specific job shop schedule problem JSSP was designed to solve see the Problem Description page.

JSSP has two different optimization algorithms:

1. Parallel Tabu Search
2. Genetic Algorithm

8.1 Features

1. Find near optimal solutions to flexible job shop schedule problems with sequence dependency setup times.
2. Use of Cython C extensions for fast execution of code.
3. Plot tabu search and/or genetic algorithm optimization using plotly.
4. Create gantt charts using plotly.
5. Create production schedule excel file.

CHAPTER 9

Indices and Tables

- genindex
- modindex
- search

Python Module Index

j

JSSP, 21
JSSP.benchmark_plotter, 26
JSSP.data, 27
JSSP.exception, 29
JSSP.genetic_algorithm, 21
JSSP.genetic_algorithm.ga, 21
JSSP.solution, 23
JSSP.solution.factory, 23
JSSP.solution.solution, 24
JSSP.solver, 29
JSSP.tabu_search, 25
JSSP.tabu_search.ts, 25

Index

C

convert_fjs_to_csv() (*JSSP.data.Data static method*), 27
create_gantt_chart_html_file() (*JSSP.solution.solution.Solution method*), 24
create_schedule_xlsx_file() (*JSSP.solution.solution.Solution method*), 24

D

Data (*class in JSSP.data*), 27

F

FITNESS_PROPORTIONATE() (*JSSP.genetic_algorithm.ga.GASelectionEnum method*), 21
FJSData (*class in JSSP.data*), 28

G

GASelectionEnum (*class in JSSP.genetic_algorithm.ga*), 21
genetic_algorithm_iter() (*JSSP.solver.Solver method*), 29
genetic_algorithm_time() (*JSSP.solver.Solver method*), 30
GeneticAlgorithmAgent (*class in JSSP.genetic_algorithm.ga*), 22
get_job() (*JSSP.data.Data method*), 27
get_job_id() (*JSSP.data.Job method*), 28
get_job_id() (*JSSP.data.Task method*), 29
get_longest_process_time_first_solution() (*JSSP.solution.factory.SolutionFactory method*), 23
get_max_sequence() (*JSSP.data.Job method*), 28
get_n_longest_process_time_first_solution() (*JSSP.solution.factory.SolutionFactory method*), 23
get_n_shortest_process_time_first_solution() (*JSSP.solution.factory.SolutionFactory method*), 23
get_n_solutions() (*JSSP.solution.factory.SolutionFactory method*), 23
get_number_of_tasks() (*JSSP.data.Job method*), 28
get_operation_list_for_machine() (*JSSP.solution.solution.Solution method*), 25
get_pieces() (*JSSP.data.Task method*), 29
get_runtime() (*JSSP.data.Data method*), 27
get_sequence() (*JSSP.data.Task method*), 29
get_setup_time() (*JSSP.data.Data method*), 27
get_shortest_process_time_first_solution() (*JSSP.solution.factory.SolutionFactory method*), 23
get_solution() (*JSSP.solution.factory.SolutionFactory method*), 23
get_task() (*JSSP.data.Job method*), 28
get_task_id() (*JSSP.data.Task method*), 29
get_tasks() (*JSSP.data.Job method*), 28
get_usable_machines() (*JSSP.data.Task method*), 29

I

IncompleteSolutionException, 29
InfeasibleSolutionException, 29
iplot_benchmark_results() (*in module JSSP.benchmark_plotter*), 26
iplot_benchmark_results() (*JSSP.solver.Solver method*), 30
iplot_gantt_chart() (*JSSP.solution.solution.Solution method*), 25

J

Job (*class in JSSP.data*), 28

job_task_index_matrix (*JSSP.data.Data attribute*), 28
jobs (*JSSP.data.Data attribute*), 28
JSSP (*module*), 21
JSSP.benchmark_plotter (*module*), 26
JSSP.data (*module*), 27
JSSP.exception (*module*), 29
JSSP.genetic_algorithm (*module*), 21
JSSP.genetic_algorithm.ga (*module*), 21
JSSP.solution (*module*), 23
JSSP.solution.factory (*module*), 23
JSSP.solution.solution (*module*), 24
JSSP.solver (*module*), 29
JSSP.tabu_search (*module*), 25
JSSP.tabu_search.ts (*module*), 25

M

machine_speeds (*JSSP.data.Data attribute*), 28

O

Operation (*class in JSSP.solution.solution*), 24
output_benchmark_results() (*in module JSSP.benchmark_plotter*), 26
output_benchmark_results() (*JSSP.solver.Solver method*), 30

R

RANDOM() (*JSSP.genetic_algorithm.ga.GASelectionEnum method*), 22

S

sequence_dependency_matrix (*JSSP.data.Data attribute*), 28
set_max_sequence() (*JSSP.data.Job method*), 28
Solution (*class in JSSP.solution.solution*), 24
SolutionFactory (*class in JSSP.solution.factory*), 23
Solver (*class in JSSP.solver*), 29
SpreadsheetData (*class in JSSP.data*), 28
start() (*JSSP.genetic_algorithm.ga.GeneticAlgorithmAgent method*), 22
start() (*JSSP.tabu_search.ts.TabuSearchAgent method*), 26

T

tabu_search_iter() (*JSSP.solver.Solver method*), 31
tabu_search_time() (*JSSP.solver.Solver method*), 31
TabuSearchAgent (*class in JSSP.tabu_search.ts*), 25
Task (*class in JSSP.data*), 29
task_processing_times_matrix (*JSSP.data.Data attribute*), 28
TaskWrapper (*class in JSSP.solution.factory*), 24
TOURNAMENT() (*JSSP.genetic_algorithm.ga.GASelectionEnum method*), 22

U

usable_machines_matrix (*JSSP.data.Data attribute*), 28