moptipy

The Metaheuristic Optimization in Python Package

`moptipy` is an open source Python software package for metaheuristic optimization algorithms available on PyPi and GitHub (https://thomasweise.github.io/moptipy). `moptipy` has the following key features, which make it suitable for scientific research, real-world industrial applications, and student projects.

green text = clickable hyperlink

1. Very comprehensive documentation with many examples, reaching down to literature references inside the code and up to complex example experiments. It is also accompanied by a free online book (at https://thomasweise.github.io/oa) on metaheuristic optimization and the implemented algorithms.
2. Several standard search spaces (bit strings, permutations, real vectors), operators, and algorithms, including randomized local search, simulated annealing, evolutionary algorithms, memetic algorithms, NSGA-II, several numerical optimization algorithms, etc., are already implemented and ready for use.
3. You can easily implement own algorithms, operators, objective functions, or search spaces.
4. You can also easily integrate algorithms from external libraries and unify them under our API, which we did as proof-of-concept with CMA-ES, BOBYQA, as well as for the algorithms from SciPy.
5. Stopping criteria for optimization processes can be defined based on goal solution qualities, clock time, and/or consumed objective function evaluations.
6. Data collection at selectable verbosity level, ranging from only providing the final result and its quality without creating any log file to creating log files with all (or all improving) steps of an algorithm, the result, algorithm and problem parameters, system setup, non-dominated solutions, and the random seed.
7. An experiment execution facility for simple and robust parallel and distributed experimentation.
8. All experiments are fully reproducible, i.e., from a log file you can configure an algorithm and problem such that *exactly* the same search steps are performed as in the original setting.
9. An experiment evaluation facility that can parse the log files and generate progress plots, result tables, ERT and ECDF plots, statistical test tables, and export data towards Excel or the popular IOHanalyzer.
10. Support for both single-objective and multi-objective optimization.
11. Good unit test coverage plus pre-defined tools to unit test your own code.
12. High code quality: Our package not just undergoes thorough unit tests, but also comprehensive static code analysis on every commit, using 19 different tools (and passing their checks).
13. The package is written in Python (>= 3.10), which currently probably is the predominant language in machine learning and AI as well as maybe the most-often used language in university classes. `moptipy` is therefore ideal for the use by both students and practitioners in AI, ML, or computer science in general.
14. Regular releases with improvements and additions on PyPi (https://pypi.org/project/moptipy).
15. Open source, with code available at https://github.com/thomasWeise/moptipy. Licensed under GPL 3.0. Terms for a special-purpose licenses can be discussed if need be (see contact information at the bottom).
16. Simple and quick installation via "`pip install moptipy`".
Obtain the source code via "`git clone https://github.com/thomasWeise/moptipy`".

Our package is designed to be particularly easy to use and to be very versatile. You can easily implement new algorithms for your specific optimization problems. It also allows for comprehensive experiments to find out which algorithm and algorithm configuration performs well for your scenario. You can collect a lot of data and evaluate it. You can then use the best algorithm setup and switch off the data collection in the final application for maximum performance. Since `moptipy` is accompanied by a free e-book, it is also suitable for students who are just beginning to step into the field of optimization. The experiment execution, data collection, and data evaluation facilities make the code useful for scientific research. Finally, due to its high code quality, comprehensive documentation and unit test facilities, it is also suitable for practical applications and actual industrial scenarios.

Contact: If you have any questions or suggestions, please contact Prof. Dr. Thomas Weise (汤卫思教授) of the Institute of Applied Optimization (应用优化研究所, IAO) of the School of Artificial Intelligence and Big Data (人工智能与大数据学院) at Hefei University (合肥学院) in Hefei, Anhui, China (中国安徽省合肥市) via email to tweise@hfuu.edu.cn, always with CC to tweise@ustc.edu.cn.

Examples for Running Experiments using `moptipy`

Here we provide two very simple examples about how to execute experiments using `moptipy`. More examples can be found at https://thomasweise.github.io/moptipy/#examples, including examples on continuous optimization with and without logging, generating ECDF plots, ERT plots, ERT-ECDF plots, end result CSV files, -plots, -tables, -statistical tests, and -statistics CSV files, plots relating end results to algorithm parameters or instance features, plots showing algorithm progress over time, an example of implementing an own algorithm and own problem, the log file structure, and an example for multi-objective optimization. *See also the next page.*

Apply One Algorithm Once to One Problem

The Code

```python
from moptipy.algorithms.so.rls import RLS
from moptipy.api.execution import Execution
from moptipy.examples.bitstrings.onemax import OneMax
from moptipy.operators.bitstrings.op0_random import Op0Random
from moptipy.operators.bitstrings.op1_flip1 import Op1Flip1
from moptipy.spaces.bitstrings import BitStrings
from moptipy.utils.temp import TempFile

space = BitStrings(10)  # search in bit strings of length 10
problem = OneMax(10)  # we maximize the number of 1 bits
algorithm = RLS(  # create RLS that
    Op0Random(),  # starts with a random bit string and
    Op1Flip1())  # flips exactly one bit in each step

# We work with a temporary log file which is automatically deleted after this
# experiment. For a real experiment, you would not use the `with` block and
# instead put the path to the file that you want to create into `tf` by doing
# `from moptipy.utils.path import Path; tf = Path.path("mydir/my_file.txt")`.
with TempFile.create() as tf:  # create temporary file `tf`
    ex = Execution()  # begin configuring execution
    ex.set_solution_space(space)  # set solution space
    ex.set_objective(problem)  # set objective function
    ex.set_algorithm(algorithm)  # set algorithm
    ex.set_rand_seed(199)  # set random seed to 199
    ex.set_log_file(tf)  # set log file = temp file `tf`
    ex.set_max_fes(100)  # allow at most 100 function evaluations
    with ex.execute() as process:  # now run the algorithm*problem combination
        end_result = process.create()  # create empty record to receive result
        process.get_copy_of_best_y(end_result)  # obtain end result
        print(f"Best solution found: {process.to_str(end_result)}")
        print(f"Quality of best solution: {process.get_best_f()}")
        print(f"Consumed Runtime: {process.get_consumed_time_millis()}ms")
        print(f"Total FEs: {process.get_consumed_fes()}")

    print("\nNow reading and printing all the logged data:")
    print(tf.read_all_str())  # instead, we load and print the log file
# The temp file is deleted as soon as we leave the `with` block.
```

The Explanation

In `moptipy`, the application of one algorithm to one problem instance can be configured via the `Execution` builder object. Here, you set the solution space, the objective function, and the algorithm. You can specify a termination criterion as well as a random seed and a log file. Invoking the `execute()` method yields a `Process` object, from which you then can query the end result.

The Output

```
Best solution found: TTTTTTTTTT
Quality of best solution: 0
Consumed Runtime: 129ms
Total FEs: 17

Now reading and printing all the logged data:
BEGIN_STATE
totalFEs: 17
totalTimeMillis: 129
bestF: 0
lastImprovementFE: 17
lastImprovementTimeMillis: 129
END_STATE
BEGIN_SETUP
...
p.maxFEs: 100
p.goalF: 0
p.randSeed: 199
...
END_SETUP
BEGIN_SYS_INFO
...
END_SYS_INFO
BEGIN_RESULT_Y
TTTTTTTTTT
END_RESULT_Y
```

Run an Experiment Applying Two Algorithms Five Times to Four Problems

The Code

```python
from moptipy.algorithms.so.rls import RLS
from moptipy.algorithms.random_sampling import RandomSampling
from moptipy.api.execution import Execution
from moptipy.api.experiment import run_experiment
from moptipy.evaluation.end_results import EndResult
from moptipy.examples.bitstrings.leadingones import LeadingOnes
from moptipy.examples.bitstrings.onemax import OneMax
from moptipy.operators.bitstrings.op0_random import Op0Random
from moptipy.operators.bitstrings.op1_flip1 import Op1Flip1
from moptipy.spaces.bitstrings import BitStrings
from moptipy.utils.temp import TempDir

# The four problems we want to try to solve:
problems = [lambda: OneMax(10),  # 10-dimensional OneMax
            lambda: OneMax(32),  # 32-dimensional OneMax
            lambda: LeadingOnes(10),  # 10-dimensional LeadingOnes
            lambda: LeadingOnes(32)]  # 32-dimensional LeadingOnes

def make_rls(problem) -> Execution:
    ex = Execution()
    ex.set_solution_space(BitStrings(problem.n))
    ex.set_objective(problem)
    ex.set_algorithm(RLS(  # create RLS that
        Op0Random(),  # starts with a random bit string and
        Op1Flip1()))  # flips one bit in each step
    ex.set_max_fes(100)  # permit 100 FEs
    return ex

def make_random_sampling(problem) -> Execution:
    ex = Execution()
    ex.set_solution_space(BitStrings(problem.n))
    ex.set_objective(problem)
    ex.set_algorithm(RandomSampling(Op0Random()))
    ex.set_max_fes(100)
    return ex

# We execute the whole experiment in a temp directory.
# For a real experiment, you would put an existing directory path in `td`
# by doing `from moptipy.utils.path import Path; td = Path.directory("mydir")`
# and not use the `with` block.
with TempDir.create() as td:  # create temporary directory `td`
    run_experiment(base_dir=td,  # set the base directory for log files
                   instances=problems,  # define the problem instances
                   setups=[make_rls,  # provide RLS run creator
                           make_random_sampling],  # provide RS run creator
                   n_runs=5)  # we will execute 5 runs per setup

    EndResult.from_logs(  # parse all log files and print end results
        td, lambda er: print(f"{er.algorithm} on {er.instance}: {er.best_f}"))
# The temp directory is deleted as soon as we leave the `with` block.
```

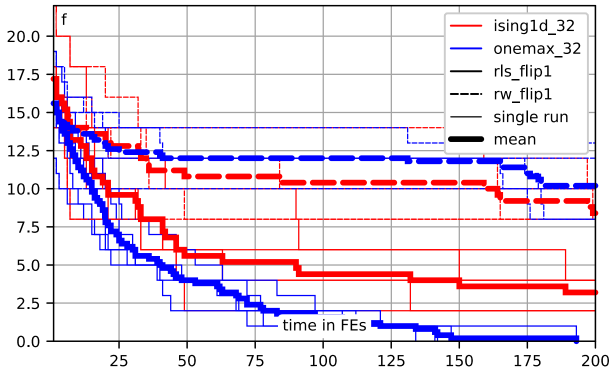
The Explanation

A replicable series of executions, i.e., the application of multiple algorithms to multiple problem instances for multiple runs, can be performed via function `run_experiment` from module `moptipy.api.experiment`. It will automatically generate a folder structure of log files that can then be parsed by the evaluation tools.

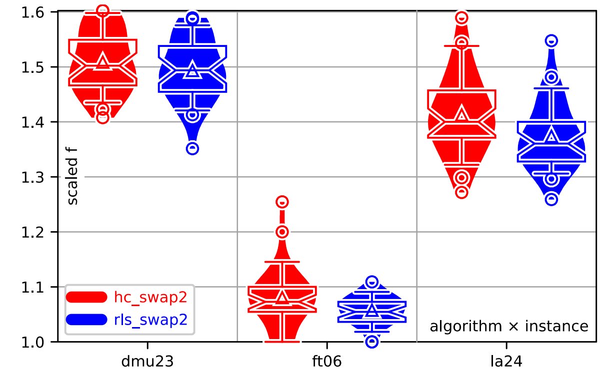
The Output

```
rs on onemax_10: 0
rs on onemax_10: 2
rs on onemax_10: 1
rs on onemax_10: 2
rs on onemax_10: 1
rs on onemax_32: 8
rs on onemax_32: 8
rs on onemax_32: 8
rs on onemax_32: 9
rs on onemax_32: 9
rs on leadingones_32: 26
rs on leadingones_32: 26
rs on leadingones_32: 25
rs on leadingones_32: 26
rs on leadingones_32: 23
rs on leadingones_10: 4
rs on leadingones_10: 0
rs on leadingones_10: 3
rs on leadingones_10: 3
rs on leadingones_10: 0
rls_flip1 on onemax_10: 0
rls_flip1 on onemax_10: 0
rls_flip1 on onemax_10: 0
rls_flip1 on onemax_10: 0
rls_flip1 on onemax_10: 0
rls_flip1 on onemax_32: 2
rls_flip1 on onemax_32: 1
rls_flip1 on onemax_32: 2
rls_flip1 on onemax_32: 2
rls_flip1 on onemax_32: 1
rls_flip1 on leadingones_32: 18
rls_flip1 on leadingones_32: 23
rls_flip1 on leadingones_32: 28
rls_flip1 on leadingones_32: 16
rls_flip1 on leadingones_32: 29
rls_flip1 on leadingones_10: 0
rls_flip1 on leadingones_10: 0
rls_flip1 on leadingones_10: 0
rls_flip1 on leadingones_10: 0
rls_flip1 on leadingones_10: 0
```

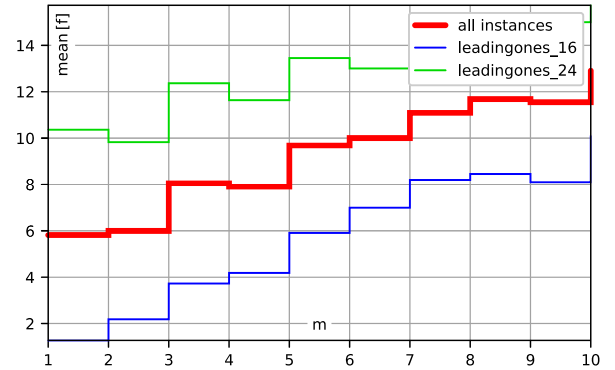
Examples for Plots Generated from Experiments



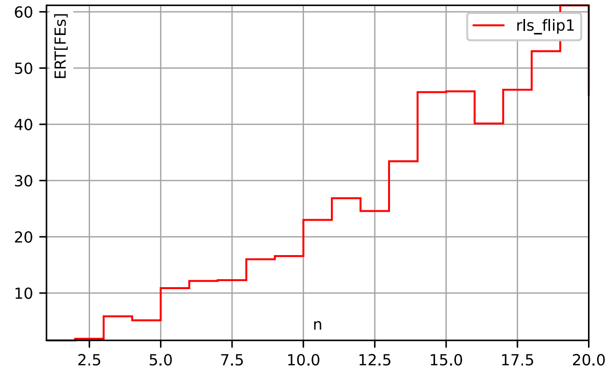
The progress of a RLS and a random walk over the consumed objective function evaluations, based on five runs per algorithm on the 32-bit OneMax and 1D-Ising model. [see this example]



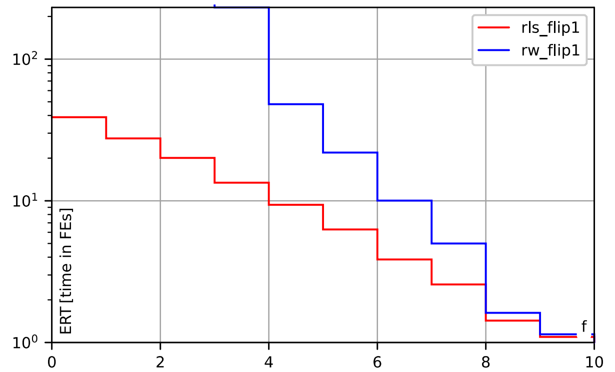
The distribution of the end results of a Hill Climber and a RLS on three instances of the Job Shop Scheduling Problem (JSSP), as box plots on top of violin plots, estimated based on 31 runs per setup. [see this example]



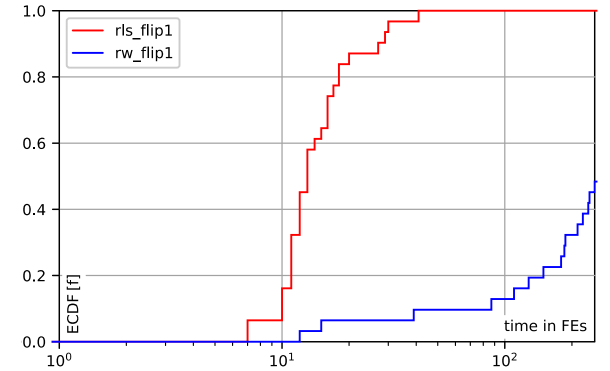
The mean end result quality reached by an RLS flipping each bit with probability $p=m/n$ on LeadingOnes problems with $n=16$ and $n=24$, estimated from 11 runs for 128 FEs plotted over the parameter m. [see this example]



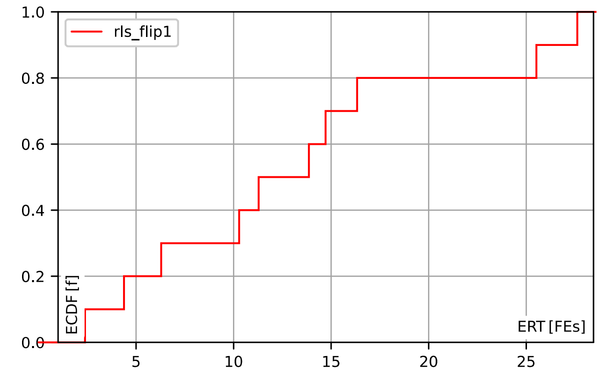
The expected running time (ERT) of 1-flip RLS in relation to the number of bits n of a OneMax problem, estimated by 7 runs per value of n. [see this example]



The expected running time (ERT, vertical axis) in relation to the required goal objective values (f, horizontal axis) for RLS and a random walk on the 16-bit OneMax with a 100 FE budget and 21 runs per algorithm. [see this example]



The empirical cumulative distribution functions (ECDF) showing the fraction of runs that successfully solved the problem over the runtime (in FEs, log-scaled) for an RLS and a random walk applied to 8-bit OneMax for 256 FEs and 31 runs. [see this example]



The fraction of problem *instances* that an 1-flip RLS can solve on the instance-based ERT for OneMax with 2 to 11 bits, estimated based on 21 runs. [see this example]

I	lb(f)	setup	best	mean	sd	mean1	mean(fes)	mean(t)
dmu23	4'668	hc_swap2	6'260	6'413.6	191.78	1.374	626	10
		rls_swap2	**5'886**	**6'177.7**	164.08	**1.323**	**704**	**11**
		rs	7'378	7'576.6	**122.78**	1.623	357	8
ft06	55	hc_swap2	57	59.3	1.25	1.078	133	2
		rls_swap2	**55**	**57.0**	1.91	**1.036**	333	4
		rs	60	60.4	**0.79**	1.099	**651**	**5**
la24	935	hc_swap2	1'122	1'180.7	61.74	1.263	**752**	9
		rls_swap2	**1'078**	**1'143.0**	48.23	**1.222**	752	**10**
		rs	1'375	1'404.3	**26.66**	1.502	248	3
		setup	best1	gmean1	worst1	sd1	mean(fes)	mean(t)
summary		hc_swap2	1.036	1.231	1.444	0.1	504	7
summary		rls_swap2	**1.000**	**1.187**	**1.377**	**0.1**	**596**	**8**
summary		rs	1.091	1.389	1.650	0.2	419	5

Tables with end results of algorithms on different (here: JSSP) instances can be generated in Markdown, LaTeX, and HTML format. [see this example]

Comprehensive Log Files for Reproducible Experiments

The Code:

```python
from moptipy.algorithms.so.rls import RLS  # the algorithm we use
from moptipy.examples.jssp.experiment import run_experiment  # the runner
from moptipy.operators.permutations.op0_shuffle import Op0Shuffle  # 0-ary op
from moptipy.operators.permutations.op1_swap2 import Op1Swap2  # 1-ary op
from moptipy.utils.temp import TempDir  # temp directory tool

# We work in a temporary directory, i.e., delete all generated files on exit.
# For a real experiment, you would put an existing directory path in `td`
# by doing `from moptipy.utils.path import Path; td = Path.directory("mydir")`
# and not use the `with` block.
with TempDir.create() as td:  # create temp directory
    # Execute an experiment consisting of exactly one run.
    # As example domain, we use the job shop scheduling problem (JSSP).
    run_experiment(
        base_dir=td,  # working directory = temp dir
        algorithms=[  # the set of algorithms to use: we use only 1
            # an algorithm is created via a lambda
            lambda inst, pwr: RLS(Op0Shuffle(pwr), Op1Swap2())],
        instances=("demo",),  # use the demo JSSP instance
        n_runs=1)  # perform exactly one run
    # The random seed is automatically generated based on the instance name.
    print(td.resolve_inside(  # so we know algorithm, instance, and seed
        "rls_swap2/demo/rls_swap2_demo_0x5a9363100a272f12.txt")
          .read_all_str())  # read file into string (which then gets printed)
# When leaving "while", the temp directory will be deleted
```

The Explanation

The basis for any research, be it purely scientific or industrial, is that experiments are clearly and comprehensively documented and reproducible and repeatable (ACM definition). Yet, this is often disregarded and only summary statistics are preserved, the actual solutions are not stored but only their quality, and so on.

If you generate log files with `moptipy`, they can automatically document your experiments and collect the relevant information to enable you or other researchers to exactly reproduce the results later on. It does so *automatically*.

Here we show a program and the log file it creates for a single-objective optimization process. The log file format for multi-objective optimization is compatible.

The Results

```
BEGIN_PROGRESS
fes;timeMS;f
1;1;267
5;1;235
10;1;230
20;1;227
25;1;205
40;1;200
84;2;180
END_PROGRESS
```

Section `PROGRESS` contains either the improving steps or all steps that an algorithm takes. It logs the consumed objective function evaluations, the runtime in milliseconds, and the best-so-far solution quality. It is optional.

```
BEGIN_STATE
totalFEs: 84
totalTimeMillis: 2
bestF: 180
lastImprovementFE: 84
lastImprovementTimeMillis: 2
END_STATE
```

Section `STATE` contains the end state of the optimization process, including the best objective value, when it was found, and how much runtime was used.

```
BEGIN_SETUP
p.name: LoggingProcessWithSearchSpace
p.class: ↩
moptipy.api._process_ss_log._ProcessSSLog
p.maxTimeMillis: 120000
p.goalF: 180
p.randSeed: 6526669205530947346
p.randSeed(hex): ↩
0x5a9363100a272f12
p.randGenType: numpy.random. ↩
_generator.Generator
p.randBitGenType: ↩
numpy.random._pcg64.PCG64
a.name: rls_swap2
a.class: moptipy.algorithms. ↩
rls.RLS
a.op0.name: shuffle
a.op0.class: ↩
moptipy. operators.permutations. op0_shuffle.Op0Shuffle
a.op1.name: swap2
a.op1.class: ↩
moptipy.operators.permutations.op1_swap2.Op1Swap2
y.name: gantt_demo
y.class: moptipy.examples.jssp.gantt_space.GanttSpace
y.shape: (5, 4, 3)
y.dtype: h
y.inst.name: demo
y.inst.class: moptipy.examples.jssp.instance.Instance
y.inst.machines: 5
y.inst.jobs: 4
y.inst.makespanLowerBound: 180
y.inst.makespanUpperBound: 482
y.inst.dtype: b
f.name: makespan
f.class: moptipy.examples.jssp.makespan.Makespan
```

Section `SETUP` contains the setup of the optimization process (p.*) with the random seed, the algorithm (a.*), the solution space (y.*), the objective function (f.*), the search space (x.*), and the encoding (g.*). It allows for recreating the exact system setup and therefore for maximally reproducible experiments.

[…continued in right column…]

[…continued from left column…]

```
x.name: perm4w5r
x.class: moptipy.spaces.permutations.Permutations
x.nvars: 20
x.dtype: b
x.min: 0
x.max: 3
x.repetitions: 5
g.name: operation_based_encoding
g.class: ↩
moptipy.examples.jssp.ob_encoding.OperationBasedEncoding
g.dtypeMachineIdx: b
g.dtypeJobIdx: b
g.dtypeJobTime: h
END_SETUP
BEGIN_SYS_INFO
session.start: 2022-05-03 08:49:14.883057+00:00
session.node: home
session.procesId: 0xc4b9
session.cpuAffinity: ↩
0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15
session.ipAddress: 192.168.1.105
version.moptipy: 0.8.5
version.numpy: 1.21.5
version.numba: 0.55.1
version.matplotlib: 3.5.1
version.psutil: 5.9.0
version.scikitlearn: 1.0.2
hardware.machine: x86_64
hardware.nPhysicalCpus: 8
hardware.nLogicalCpus: 16
hardware.cpuMhz: (2200MHz..3700MHz)*16
hardware.byteOrder: little
hardware.cpu: AMD Ryzen 7 2700X Eight-Core Processor
hardware.memSize: 16719478784
python.version: 3.10.4 (main, Apr 2 2022) [GCC 11.2.0]
python.implementation: CPython
os.name: Linux
os.release: 5.15.0-27-generic
os.version: 28-Ubuntu SMP Thu Apr 14 04:55:28 UTC 2022
END_SYS_INFO
```

Section `SYS_INFO` contains the system configuration, such as the OS, CPU, GPU, the versions of Python, `moptipy`, and the libraries it depends on, etc. This allows for reproducing the system environment if need be.

```
BEGIN_RESULT_Y
1;20;30;0;30;40;3;145;165;2;170;180;1;0;20;0;40;60;2; ↩
60;80;3;165;180;2;0;30;0;60;80;1;80;130;3;130;145;1; ↩
30;60;3;60;90;0;90;130;2;130;170;3;0;50;2;80;92;1;130; ↩
160;0;160;170
END_RESULT_Y
BEGIN_RESULT_X
2;1;3;1;0;0;2; ↩
0;1;2;3;1;0;2; ↩
1;3;0;3;2;3
END_RESULT_X
```

The `END_RESULT_*` sections contain the textual representation of the best solution discovered (Y) and, if search and solution space are different, the matching point in the search space (X).