QMKPy

Release 1.2.0

Karl-Ludwig Besser

Oct 25, 2022

CONTENTS:

QMKPy is a Python library for modeling and solving quadratic multiple knapsack problems (QMKP). It provides a framework that allows quickly implementing and testing novel algorithms to solve the QMKP. It is therefore primarily targeting researchers working in the area of operations research/optimization.

Additionally, it can be used to easily generate datasets of QMKP instances which can be used as reference test set to fairly compare different algorithms.

The QMKP is an assignment problem where $N \in \mathbb{N}$ items are assigned to $K \in \mathbb{N}$ knapsacks such that an overall profit is maximized. The exact formulation of the QMKP that can be solved by this package is given as follows

$$\max \quad \sum_{u \in \mathcal{K}} \left(\sum_{i \in \mathcal{A}(u)} p_i + \sum_{\substack{j \in \mathcal{A}(u) \\ j \neq i}} p_{ij} \right)$$

$$\text{s.t.} \quad \sum_{i \in \mathcal{A}(u)} w_i \leq c_u \qquad \forall u \in \mathcal{K} \tag{2}$$

$$\sum_{u=1}^{K} a_{iu} \leq 1 \qquad \forall i \in \{1, 2, \ldots, N\} \tag{3}$$

where $\mathcal{K} = \{1, 2, \ldots, K\}$ describes the set of K knapsacks, $\mathcal{A}(u)$ is the set of items that are assigned to knapsack u, and $a_{iu} \in \{0, 1\}$ is the indicator whether item i is assigned to knapsack u. Each item i has the weight $w_i \in \mathbb{R}_+$ and knapsack u has the weight capacity $c_u \in \mathbb{R}_+$. When assigning item i to a knapsack, it yields the non-negative profit $p_i \in \mathbb{R}_+$. When assigning item j (with $j \neq i$) to the same knapsack, the additional (joint) profit $p_{ij} \in \mathbb{R}_+$ is obtained.

The objective of the above optimization problem is to maximize the total profit such that each item is assigned to at most one knapsack and such that the weight capacity constraints of the knapsacks are not violated.

Remark: The profits p are also referred to as "values" in the literature.

A detailed description of the way how the mathematical components of the QMKP are implemented in the `qmkpy` framework can be found in the code conventions page.

For a basic overview on knapsack problems, see [KPP04].

CHAPTER

ONE

INSTALLATION

The easiest way to install the package is by using pip

```
pip3 install qmkpy
```

You can also install the latest version from the Github repository

```
git clone https://github.com/klb2/qmkpy
cd qmkpy
pip3 install -r requirements.txt
pip3 install .
```

CHAPTER

TWO

GETTING STARTED

In the following, a few simple examples are shown.

2.1 Basic Usage

The following script contains an example in which a QMKP is defined and solved by the implemented constructive procedure.

Listing 1: Defining and Solving a QMKP

```
import numpy as np
from qmkpy import total_profit_qmkp, QMKProblem
from qmkpy import algorithms

weights = [5, 2, 3, 4]  # four items
capacities = [10, 5, 12, 4, 2]  # five knapsacks
profits = np.array([[3, 1, 0, 2],
                    [1, 1, 1, 4],
                    [0, 1, 2, 2],
                    [2, 4, 2, 3]])  # symmetric profit matrix

qmkp = QMKProblem(profits, weights, capacities)
qmkp.algorithm = algorithms.constructive_procedure
assignments, total_profit = qmkp.solve()

print(assignments)
print(total_profit)
```

2.2 Saving a Problem Instance

It is possible to save a problem instance of a QMKP. This can be useful to share examples as a benchmark dataset to compare different algorithms.

Listing 2: Saving a QMKProblem Instance

```
import numpy as np
from qmkpy import QMKProblem

```

(continues on next page)

(continued from previous page)

```
weights = [5, 2, 3, 4]  # four items
capacities = [10, 5, 12, 4, 2]  # five knapsacks
profits = np.array([[3, 1, 0, 2],
                    [1, 1, 1, 4],
                    [0, 1, 2, 2],
                    [2, 4, 2, 3]])  # symmetric profit matrix

qmkp = QMKProblem(profits, weights, capacities)

# Save the problem instance using the Numpy npz format
qmkp.save("my_problem.npz", strategy="numpy")
```

2.3 Loading a Saved QMKProblem Instance

You can also load a previously saved QMKProblem instance to get a *qmkpy.QMKProblem* object with the same profits, weights and weight capacities.

Listing 3: Loading a Saved QMKProblem

```
from qmkpy import QMKProblem

saved_problem = "my_problem.npz"  # saved model file
qmkp = QMKProblem.load(saved_problem, strategy="numpy")
```

CHAPTER

THREE

CODE CONVENTIONS

The code in the library follows some conventions, which are specified in the following. We assume that there are N items and K knapsacks.

3.1 Basic Arrays

In the following, we will discuss the essential elements of the QMKP and their implementation in the `qmkpy` framework.

3.1.1 Mathematical Description

The four essential components of the QMKP are the following.

Profit matrix $P \in \mathbb{R}_+^{N \times N}$
: This symmetric matrix contains the profit values p_i on the main diagonal and the joint profit values p_{ij} as the other elements.

Weights $w \in \mathbb{R}_+^N$
: This vector contains the weights of the items, where the i-th component w_i corresponds to the weight of item i.

Capacities $c \in \mathbb{R}_+^K$
: This vector contains the capacities of the knapsacks, where the i-th component c_i corresponds to the capacity of knapsack i.

Assignments $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K\}$ **with** $\mathcal{A}_i \subseteq \{1, 2, \ldots, N\}$
: The assignments of items to knapsacks are collected in the set $\mathcal{A}$. It contains the individual sets $\mathcal{A}_i$ which contains the indices of all items that are assigned to knapsack i.

3.1.2 Implementation

The three main components described above are implemented in `qmkpy` as arrays. The details are as follows.

`profits`
: The profit matrix is implemented as an array of size `[N, N]` which represents the symmetric $N \times N$ matrix P. We have that the profits of the individual items p_i are placed on the main diagonal `profits[i-1, i-1] = p_i` and the joint profits p_{ij} make up the other elements as `profits[i-1, j-1] = profits[j-1, i-1] = p_{ij}`. (The `-1` index shift is due to Python's 0-based indexing.)

`weights`
: The weight vector is implemented as a list of length `N`, where the weight w_i corresponds to the index `i-1`, i.e., `weights[i-1] = w_i`.

capacities
: The capacities vector is implemented as a list of length `K`, where the capacity c_i corresponds to the index `i-1`, i.e., `capacities[i-1] = c_i`.

assignments
: There are multiple ways of representing the assignment of items to knapsacks. *For all algorithms, the binary representation is used to represent the solution to a QMKP.* In this, the assignments $\mathcal{A}$ are represented by a binary array of size `[N, K]` where row i stands for item i and column u represents knapsack u. Thus, element `assignments[i-1, u-1] = 1`, if item i is assigned to knapsack u and `assignments[i-1, u-1] = 0` otherwise.

3.2 Argument Order

Functions that work on a QMKP always assume the argument order `profits, weights, capacities` and they are expected to return `assignments` in the binary form described above.

So if you want to write a function that solves a QMKP, the argument list of your function needs to start with this. More details on this can also be found on the Implementing a Novel Algorithm page.

3.3 Alternative Representation of the Assignment Matrix

There are multiple ways of representing the final solution to a QMKP. Essentially, we need to represent the assignment of the items to the knapsacks.

Besides the binary representation of the algorithms, which is described above, another popular representation is the chromosome form $C \in \{0, 1, \ldots, K\}^N$ which is a vector of length N, where the value of entry i specifies the knapsack to which item i is assigned. If the item is not assigned to any knapsack, the value 0 is used. In the `qmkpy` framework, this is implemented such that `chromosome` is a list of length `N`, where index `i-1` represents item i, i.e., `chromosome[i-1] = u-1` indicates that item i is assigned to knapsack u. If item i is not assigned to any knapsack, we have `chromosome[i-1] = -1`.

While the binary representation is dominantly used in this library, there exist functions to convert the to representations (see *qmkpy.util.assignment_from_chromosome()* and *qmkpy.util.chromosome_from_assignment()*).

CHAPTER

FOUR

IMPLEMENTING A NOVEL ALGORITHM

Note: **TL;DR:** Your function needs to be callable as: `func(profits, weights, capacities, *args)` and needs to return `assignments` in the binary form.

If you want to implement and test a novel solution algorithm for the QMKP, you simply need to write a Python function that takes `profits` as first argument, `weights` as second, and `capacities` as third argument. Beyond that, it can have an arbitrary number of additional arguments. However, it needs to be possible to pass them positionally.

The return of the function needs to be the assignment matrix in binary form.

The following example is also illustrated in a Jupyter notebook that you can either run locally or using an online service like Binder.

4.1 Example

As an example, we want to implement the following algorithm

> Assign the item i with the smallest weight w_i to the first knapsack k where it fits, i.e., where $c_k \geq w_i$.

Obviously, this algorithm ignores the profits and will not yield very good results. However, it only serves demonstration purposes.

4.1.1 Algorithm Implementation

The above algorithm could be implemented as follows

Listing 1: Example Algorithm

```
def example_algorithm(profits, weights, capacities):
    assignments = np.zeros((len(weights), len(capacities)))
    remaining_capacities = np.copy(capacities)
    items_by_weight = np.argsort(weights)
    for _item in items_by_weight:
        _weight = weights[_item]
        _first_ks = np.argmax(remaining_capacities >= _weight)
        assignments[_item, _first_ks] = 1
        remaining_capacities[_first_ks] -= _weight
    return assignments
```

It should be emphasized that you should **not** modify any of the input arrays, e.g., `capacities` inplace, since this could lead to unintended consequences.

4.1.2 Using the Algorithm

The newly implemented algorithm can then easily be used as follows.

Listing 2: Testing the Novel Algorithm

```
import numpy as np
from qmkpy import total_profit_qmkp, QMKProblem
from qmkpy import algorithms

weights = [5, 2, 3, 4]  # four items
capacities = [1, 5, 5, 6, 2]  # five knapsacks
profits = np.array([[3, 1, 0, 2],
                    [1, 1, 1, 4],
                    [0, 1, 2, 2],
                    [2, 4, 2, 3]])  # symmetric profit matrix

qmkp = QMKProblem(profits, weights, capacities)
qmkp.algorithm = example_algorithm
assignments, total_profit = qmkp.solve()

print(assignments)
print(total_profit)
```

4.2 Contributing a New Algorithm to the Package

When you feel that your algorithm should be added to the QMKPy package, please follow the following steps:

1. Place your code in the *qmkpy.algorithms* module, i.e., in the `qmkpy/algorithms.py` file.
2. Make sure that you added documentation in form of a docstring. This should also include possible references to literature, if the algorithm is taken from any published work.
3. Make sure that all unit tests pass. In order to do this, add your algorithm to the `SOLVERS` list in the test file `tests/test_algorithms.py`. Additionally, you should create a new test file `tests/test_algorithm_<your_algo>.py` which includes tests that are specific to your algorithm, e.g., testing different parameter constellations. You can run all tests using the `pytest` command.

CHAPTER

FIVE

DATASETS

One of the major benefits of this package is the possibility to quickly and easily generate datasets of reference problems and test your algorithms against (existing) datasets.

Especially when benchmarking your novel algorithm against commonly used reference datasets, this will allow a simple reproducibility. A collection of some reference datasets can be found at https://github.com/klb2/qmkpy-datasets.

In the following, an example of how a repository for a research paper could look like, is presented.

5.1 Research Paper Repository

The file structure can be as simple as shown in the following.

```
project
├── dataset/
│   ├── problem1.txt
│   ├── problem2.txt
│   └── ...
└── my_algorithm.py
```

The directory `dataset/` contains all problem instances of the reference dataset, which are saved by one of the functions in *qmkpy.io*.

The file `my_algorithm.py` contains the implementation of your algorithm. It could look something like the following. Details on how to implement new algorithms can also be found on the Implementing a Novel Algorithm page.

```python
import os
import numpy as np
import qmkpy

def my_algorithm(profits, weights, capacities):
    # DOING SOME STUFF
    return assignments

def main():
    results = []
    for root, dirnames, filenames in os.walk("dataset"):
        for problem in filenames:
            qmkp = qmkpy.QMKProblem.load(problem, strategy="txt")
            qmkp.algorithm = my_algorithm
            solution, profit = qmkp.solve()
```

(continues on next page)

(continued from previous page)

```
            results.append(profit)
    print(f"Average profit: {np.mean(results):.2f}")

if __name__ == "__main__":
    main()
```

This simple script solves all problems of the dataset using your algorithm and prints the average total profit at the end.

CHAPTER

SIX

QMKPY PACKAGE

6.1 Submodules

6.1.1 qmkpy.algorithms module

Solution algorithms for the QMKP.

This module contains all the algorithms that can be used to solve the quadratic multiple knapsack problem (QMKP).

qmkpy.algorithms.**constructive_procedure**(*profits: array*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *starting_assignment: Optional[array] = None*) → array

Constructive procedure that completes a starting assignment

This constructive procedure is based on Algorithm 1 from [AGH22] and the greedy heuristic in [HJ06]. It is a greedy algorithm that completes a partial solution of the QMKP.

Parameters

- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}.
- **weights** (list of float) – List of weights w_i of the N items that can be assigned.
- **capacities** (list of float) – Capacities of the knapsacks. The number of knapsacks K is determined as K=len(capacities).
- **starting_assignments** (np.array, *optional*) – Binary matrix of size $N \times K$ which represents existing starting assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. These assignments are not modified and will only be completed. If it is *None*, no existing assignment is assumed.

Returns

assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type

np.array

Raises

ValueError – Raises a ValueError if the starting assignment is infeasible.

qmkpy.algorithms.**fcs_procedure**(*profits: array*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *alpha: Optional[float] = None*, *len_history: int = 50*) → array

Implementation of the fix and complete solution (FCS) procedure

This fix and complete solution (FCS) procedure is based on Algorithm 2 from [AGH22]. It is basically a stochastic hill-climber wrapper around the constructive procedure *constructive_procedure()* (also see [HJ06]).

Parameters

- **profits** (`np.array`) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}.
- **weights** (`list` of `float`) – List of weights w_i of the N items that can be assigned.
- **capacities** (`list` of `float`) – Capacities of the knapsacks. The number of knapsacks K is determined as `K=len(capacities)`.
- **alpha** (`float`, *optional*) – Float between 0 and 1 that indicates the ratio of assignments that should be dropped in an iteration. If not provided, a uniformly random value is chosen.
- **len_history** (`int`, *optional*) – Number of consecutive iterations without any improvement before the algorithm terminates.

Returns

assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type

`np.array`

qmkpy.algorithms.**random_assignment**(*profits: array*, *weights: Iterable[float]*, *capacities: Iterable[float]*) → array

Generate a random (feasible) assignment

This function generates a random feasible solution to the specified QMKP. The algorithm works as follows

1. Generate a random permutation of the items
2. For each item i do
    1. Determine the possible knapsacks $\mathcal{K}_i$ that could support the item
    2. Random and uniformly select a choice from $\mathcal{K}_i \cup \{\text{skip}\}$.

This way, a feasible solution is generated without the guarantee that every item is assigned (even if it could still be assigned).

Parameters

- **profits** (`np.array`) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.
- **weights** (`list`) – List of weights w_i of the N items that can be assigned.
- **capacities** (`list`) – Capacities of the knapsacks. The number of knapsacks K is determined as `K=len(capacities)`.

Returns

assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type

`np.array`

qmkpy.algorithms.**round_robin**(*profits: array*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *starting_assignment: Optional[array] = None*, *order_ks: Optional[Iterable[int]] = None*) → array

Simple round-robin algorithm

This algorithm follows a simple round-robin scheme to assign items to knapsacks. The knapsacks are iterated in the order provided by `order_ks`. In each round, the current knapsack selects the item with the highest value density that still fits in the knapsack.

Parameters

- **profits** (`np.array`) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}.
- **weights** (`list` of `float`) – List of weights w_i of the N items that can be assigned.
- **capacities** (`list` of `float`) – Capacities of the knapsacks. The number of knapsacks K is determined as `K=len(capacities)`.
- **starting_assignments** (`np.array`, *optional*) – Binary matrix of size $N \times K$ which represents existing starting assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. These assignments are not modified and will only be completed. If it is *None*, no existing assignment is assumed.
- **order_ks** (`list` of `int`, *optional*) – Order in which the knapsacks select the items. If none is given, they are iterated by index, i.e., `order_ks = range(num_ks)`.

Returns

assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type

`np.array`

6.1.2 qmkpy.checks module

Various checks/verification functions.

This module contains various functions to perform check/verify provided parameters in the context of the QMKP. For example, this includes a check whether a provided assignment complies with the weight/capacity constraints.

`qmkpy.checks.check_dimensions`(*profits: array*, *weights: Optional[Iterable[float]] = None*) → NoReturn

Simple check whether the dimensions of the parameters match.

This function checks that

1. The profit matrix is quadratic of size N,
2. The number of items is equal to N, i.e., `len(weights)==N`.

Parameters

- **profits** (`np.array`) – Symmetric matrix of size $N \times N$ containing the profits p_{ij}.
- **weights** (`list` of `float`, *optional*) – List which contains the weights of the N items.

Raises

ValueError – This function raises a `ValueError`, if there is a mismatch.

`qmkpy.checks.is_binary`(*x: Iterable[float]*) → bool

Check whether a provided array is binary

This function checks that all elements of the input are either 0 or 1.

Parameters
x (Iterable) – Array of numbers

Returns
binary – Returns True when the array x is binary and False otherwise.

Return type
bool

qmkpy.checks.**is_feasible_solution**(*assignments: array*, *profits: array*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *raise_error: bool = False*) → bool

Check whether a provided assignment is a feasible solution.

This function performs a formal check whether the provided assignments is a feasible solution of the specified QMKProblem. This means that the shapes of the arrays match and that no weight capacity constraint is violated.

Parameters

- **assignments** (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.
- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.
- **weights** (list) – List of weights w_i of the N items that can be assigned.
- **capacities** (list) – Capacities of the knapsacks. The number of knapsacks K is determined as K=len(capacities).
- **raise_error** (bool, *optional*) – If raise_error is False, the function returns a bool, that states whether the solution is feasible. If raise_error is True, the function raises a ValueError instead.

Returns
Indication if the solution is feasible (True) or not (False)

Return type
bool

Raises
ValueError – This is only raised when raise_error is True.

qmkpy.checks.**is_symmetric_profits**(*profits: array*, *raise_error: bool = False*) → bool

Check whether the profit matrix is symmetric.

This function performs a check whether the profit matrix P is symmetric. This is expected for the QMKP.

By default, the function returns True if the matrix is symmetric and False otherwise. When raise_error is set to True, a ValueError is raised instead.

Parameters

- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.
- **raise_error** (bool, *optional*) – If raise_error is False, the function returns a bool, that states whether the solution is feasible. If raise_error is True, the function raises a ValueError instead.

Returns
Indication if the solution is feasible (True) or not (False)

Return type
bool

Raises
ValueError – This is raised when raise_error is True and the matrix is not symmetric. It can also be raised when the provided profits is not a square matrix.

6.1.3 qmkpy.io module

Input/Output functions.

This module contains functions to save and load QMKP instances.

qmkpy.io.**load_problem_json**(*fname: str*)

Load a previously stored QMKProblem instance from the JSON format

This function allows loading a QMKProblem from a .json file, which was created by the *qmkpy.io.save_problem_json()* method.

See also:

qmkpy.io.save_problem_json()
For saving a model in the JSON format.

Parameters
fname (str or PathLike) – Filepath of the saved model

Returns
problem – Loaded problem instance

Return type
qmkpy.QMKProblem

qmkpy.io.**load_problem_numpy**(*fname: str*)

Load a previously stored QMKProblem instance from the Numpy format

This function allows loading a QMKProblem from a compressed .npz file, which was created by the *qmkpy.io.save_problem_numpy()* method.

See also:

qmkpy.io.save_problem_numpy()
For saving a model in the Numpy format.

numpy.load()
For details on loading the .npz format.

Parameters
fname (str or PathLike) – Filepath of the saved model

Returns
problem – Loaded problem instance

Return type
qmkpy.QMKProblem

qmkpy.io.**load_problem_pickle**(*fname: Union[str, bytes, PathLike]*)

Load a previously stored QMKProblem instance from the Pickle format

This function allows loading a QMKProblem object from a Python pickled object file.

Caution: All warnings as for the regular `pickle.load()` apply!

See also:

qmkpy.io.save_problem_pickle()
For saving a model in the Pickle format.

pickle.load()
For details on loading a pickled object.

Parameters
fname (str or PathLike) – Filepath of the saved model

Returns
problem – Loaded problem instance

Return type
qmkpy.QMKProblem

qmkpy.io.**load_problem_txt**(*fname: Union[str, bytes, PathLike]*, *sep: str = '\t'*)

Load a previously stored QMKProblem instance from the text format

This function loads a QMKProblem instance from a text file according to the format specified in *qmkpy.io.save_problem_txt()*.

See also:

qmkpy.io.save_problem_txt()
For saving a model in the text format.

Parameters

- **fname** (str or PathLike) – Filepath of the saved model
- **sep** (str) – Separator string that is used to separate the numbers in the file.

Returns
problem – Loaded problem instance

Return type
qmkpy.QMKProblem

qmkpy.io.**save_problem_json**(*fname: Union[str, bytes, PathLike]*, *problem*, *name: Optional[str] = None*)

Save a QMKProblem as a JSON file

Save a QMKProblem instance using the JavaScript Object Notation (JSON) format. This only saves the `problem.profits`, `problem.weights` and `problem.capacities` arrays, and the `problem.name` attribute if it is set.

See also:

load_problem_json()
For loading a saved model.

Parameters

- **fname** (str or PathLike) – Filepath of the model to be saved at
- **problem** (*qmkpy.QMKProblem*) – Problem instance to be saved
- **name** (str ,optional) – Optional name of the problem that is used as the first line of the output file. If it is None, it will first be checked whether the attribute problem.name is set. If this is also None, the name defaults to qmkp_{num_items:d}_{num_ks:d}_{np.random.randint(0, 1000):03d}.

Return type
None

qmkpy.io.**save_problem_numpy**(*fname: Union[str, bytes, PathLike]*, *problem*)

Save a QMKProblem using Numpys npz format

Save a QMKProblem instance using the compressed npz format. This only saves the problem.profits, problem.weights, and problem.capacities arrays.

See also:

load_problem_numpy()
For loading a saved model.

numpy.savez_compressed()
For details on the .npz format.

Parameters

- **fname** (str or PathLike) – Filepath of the model to be saved at
- **problem** (*qmkpy.QMKProblem*) – Problem instance to be saved

Return type
None

qmkpy.io.**save_problem_pickle**(*fname: Union[str, bytes, PathLike]*, *problem*)

Save a QMKProblem using the Python Pickle format

Save a QMKProblem object using the Python pickle library. By this, the whole object is stored in a binary format.

See also:

qmkpy.io.load_problem_pickle()
For loading a saved model.

pickle.dump()
For details on the underlying pickling function.

Parameters

- **fname** (str or PathLike) – Filepath of the model to be saved at
- **problem** (*qmkpy.QMKProblem*) – Problem instance to be saved

Return type
None

qmkpy.io.**save_problem_txt**(*fname: Union[str, bytes, PathLike]*, *qmkp*, *sep: str = '\t'*, *name: Optional[str] = None*)

Save a QMKProblem instance in text form

Save a QMKProblem instance in text form inspired by the format established by Alain Billionnet and Eric Soutif for the regular QKP. The original description can be found at https://cedric.cnam.fr/~soutif/QKP/format.html.

The file format is as follows:

1. The first line provides a name/reference of the problem
2. The second line specifies the number of items
3. The third line specifies the number of knapsacks
4. The fourth line is blank to separate the meta information from the rest
5. The fifth line contains the linear profits (main diagonal elements of the profit matrix) separated by `sep`.
6. The next lines contain the upper triangular part of the profit matrix (i.e., p_{ij}).
7. Blank line separating profits from the rest
8. Weights w_i of the items, separated by `sep`.
9. Blank line separating weights and capacities
10. Capacities c_u of the knapsacks, separated by `sep`.

For the example with parameters

$$P = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{pmatrix}, \quad w = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}, \quad c = \begin{pmatrix} 5 \\ 8 \\ 1 \\ 9 \\ 2 \end{pmatrix}$$

the output-file looks as follows

```
Name of the Problem
3
5

1   4   6
2   3
5

10  20  30

5   8   1   9   2
```

See also:

qmkpy.io.load_problem_txt()
For loading a saved model.

Parameters

- **fname** (`str` or `PathLike`) – Filepath of the model to be saved at
- **problem** (*qmkpy.QMKProblem*) – Problem instance to be saved

- **sep** (str) – Separator string that is used to separate the numbers in the file.
- **name** (str ,optional) – Optional name of the problem that is used as the first line of the output file. If it is None, it will first be checked whether the attribute problem.name is set. If this is also None, the name defaults to qmkp_{num_items:d}_{num_ks:d}_{np.random.randint(0, 1000):03d}.

Return type
None

6.1.4 qmkpy.qmkp module

General definitions of the quadratic multiple knapsack problem.

This module contains the basic implementation of the quadratic multiple knapsack problem (QMKP). In particular, this includes the base class *QMKProblem*.

class qmkpy.qmkp.**QMKProblem**(*profits: Union[array, Iterable[Iterable]]*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *algorithm: Optional[Callable] = None*, *args: Optional[tuple] = None*, *assignments: Optional[array] = None*, *name: Optional[str] = None*)

Bases: object

Base class to represent a quadratic multiple knapsack problem.

This class defines a standard QMKP with N items and K knapsacks.

profits

Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.

Type
np.array

weights

List of weights w_i of the N items that can be assigned.

Type
list of float

capacities

Capacities of the knapsacks. The number of knapsacks K is determined as K=len(capacities).

Type
list of float

algorithm

Function that is used to solve the QMKP. It needs to follow the argument order algorithm(profits, weights, capacities, ...).

Type
Callable, *optional*

args

Optional tuple of additional arguments that are passed to *algorithm*.

Type
tuple, *optional*

assignments

Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. This attribute is overwritten when calling *solve()*.

Type
np.array, *optional*

name

Optional name of the problem instance

Type
str, *optional*

classmethod load(*fname: str*, *strategy: str = 'numpy'*)

Load a QMKProblem instance

This functions allows loading a previously saved QMKProblem instance. The *save()* method provides a way of saving a problem.

See also:

save()
Method to save a QMKProblem instance which can then be loaded.

Parameters

- **fname** (str) – Filepath of the saved model
- **strategy** (str) – Strategy that is used to store the model. Valid choices are (case-insensitive):
  - numpy: Save the individual arrays of the model using the np.savez_compressed() function.
  - pickle: Save the whole object using Pythons pickle module
  - txt: Save the arrays of the model using the text-based format established by Billionnet and Soutif.
  - json: Save the arrays of the model using the JSON format.

Returns
problem – Loaded problem instance

Return type
QMKProblem

save(*fname: Union[str, bytes, PathLike]*, *strategy: str = 'numpy'*) → NoReturn

Save the QMKP instance

Save the profits, weights, and capacities of the problem. There exist different strategies that are explained in the strategy parameter.

See also:

load()
For loading a saved model.

Parameters

- **fname** (str or PathLike) – Filepath of the model to be saved at

- **strategy** (str) – Strategy that is used to store the model. Valid choices are (case-insensitive):
  - numpy: Save the individual arrays of the model using the np.savez_compressed() function. See also *qmkpy.io.save_problem_numpy()*.
  - pickle: Save the whole object using Pythons pickle module. See also *qmkpy.io.save_problem_pickle()*.
  - txt: Save the arrays of the model using the text-based format established by Billionnet and Soutif. See also *qmkpy.io.save_problem_txt()*.
  - json: Save the arrays of the model using the JSON format.

Return type
None

solve(*algorithm: Optional[Callable] = None*, *args: Optional[tuple] = None*) → Tuple[array, float]

Solve the QMKP

Solve the QMKP using algorithm. This function both returns the optimal assignment and the total resulting profit. This method also automatically sets the solution to the object's attribute *assignments*.

Parameters

- **algorithm** (Callable, *optional*) – Function that is used to solve the QMKP. It needs to follow the argument order algorithm(profits, weights, capacities, ...). If it is None, the object attribute *algorithm* is used.
- **args** (tuple, *optional*) – Optional tuple of additional arguments that are passed to algorithm. If it is None, the object attribute *args* is used.

Returns

- **assignments** (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.
- **total_profit** (float) – Final total profit for the found solution.

qmkpy.qmkp.**total_profit_qmkp**(*profits: array*, *assignments: array*) → float

Calculate the total profit for given assignments.

This function calculates the total profit of a QMKP for a given profit matrix P and assignments $\mathcal{A}$ as

$$\sum_{u=1}^{K} \left(\sum_{i \in \mathcal{A}_u} p_i + \sum_{\substack{j \in \mathcal{A}_u \\ j \neq i}} p_{ij} \right)$$

where $\mathcal{A}_u$ is the set of items that are assigned to knapsack u.

Parameters

- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.
- **assignments** (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Returns
Value of the total profit

Return type
`float`

6.1.5 qmkpy.util module

Utility functions.

This module contains various utility functions, e.g., the conversion from the binary assignment matrix to the chromosome form.

`qmkpy.util.`**`assignment_from_chromosome`**(*chromosome: Iterable[int]*, *num_ks: int*) → array

Return the assignment matrix from a chromosome

Return the binary assignment matrix that corresponds to the chromosome. For more details about the connection between assignment matrix and chromosome check *chromosome_from_assignment()*.

See also:

chromosome_from_assignment()
For more details on the connection between assignment matrix and chromosome.

Parameters

- **chromosome** (`np.array` or `list` of `int`) – Chromosome version of `assignments`, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.
- **num_ks** (`int`) – Number of knapsacks K.

Returns
assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type
`np.array`

`qmkpy.util.`**`chromosome_from_assignment`**(*assignments: array*) → Iterable[int]

Return the chromosome from an assignment matrix

The chromosome version of `assignments` is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Example

Assume that we have 4 items and 3 knapsacks. Let Items 1 and 4 be assigned to Knapsack 1, Item 2 is assigned to Knapsack 3 and Item 3 is not assigned. In the binary representation, this is

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

The corresponding chromosome is

$$C(A) = \begin{pmatrix} 1 & 3 & 0 & 1 \end{pmatrix}$$

However, in the 0-index based representation in Python, this function will return

```
chromosome_from_assignment(A) = [0, 2, -1, 0]
```

as the chromosome.

Parameters
assignments (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Returns
chromosome – Chromosome version of assignments, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Return type
np.array

qmkpy.util.**get_empty_knapsacks**(*assignments: Union[array, Iterable[int]]*, *num_ks: Optional[int] = None*) → Iterable[int]

Return the list of empty knapsacks

Return the list of empty knapsacks (as their indices) from a given assignment. An empty knapsack is one without any assigned item. The assignments can be either in the binary matrix form or in the chromosome form. If the chromosome form is used, the total number of knapsacks needs to be additionally provided.

Parameters

- **assignments** (np.array or list of int) – Either a binary matrix of size $N \times K$ which represents the assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. Or assignments in the chromosome form, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.
- **num_ks** (int (optional)) – Total number K of knapsacks. This only needs to provided, if the assignments are given in the chromosome form.

Returns
empty_ks – List of the indices of the empty knapsacks.

Return type
list of int

qmkpy.util.**get_remaining_capacities**(*weights: Iterable[float]*, *capacities: Iterable[float]*, *assignments: Union[array, Iterable[int]]*)

Return the remaining weight capacities of the knapsacks

Returns the remaining weight capacities of the knapsacks for given assignments. The function does *not* raise an error when a weight constraint is violated but will return a negative remaining capacity in this case.

Parameters

- **weights** (list of float) – List of weights w_i of the N items that can be assigned.
- **capacities** (list of float) – Capacities of the knapsacks. The number of knapsacks K is determined as K=len(capacities).
- **assignments** (np.array or list of int) – Either a binary matrix of size $N \times K$ which represents the assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. Or assignments in the chromosome form, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Returns
remaining_capacities – List of the remaining capacities. Can be negative, if a knapsack is overloaded.

Return type
list of float

qmkpy.util.**get_unassigned_items**(*assignments: Union[array, Iterable[int]]*) → Iterable[int]

Return the list of unassigned items

Return the list of unassigned items (as their indices) from a given assignment. It can be either in the binary matrix form or in the chromosome form.

Parameters
assignments (np.array or list of int) – Either a binary matrix of size $N \times K$ which represents the assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. Or assignments in the chromosome form, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Returns
unassigned_items – List of the indices of the unassigned items.

Return type
list of int

qmkpy.util.**value_density**(*profits: array*, *weights: Iterable[float]*, *assignments: Union[array, Iterable[int]]*, *reduced_output: bool = False*) → Iterable[float]

Calculate the value density given a set of selected objects.

This function calculates the value density of item i for knapsack k and given assignments $\mathcal{A}_k$ according to

$$\mathrm{vd}_i(\mathcal{A}_k) = \frac{1}{w_i}\left(p_i + \sum_{\substack{j \in \mathcal{A}_k \\ j \neq i}} p_{ij}\right)$$

This value indicates the profit (per item weight) that is gained by adding the item i to knapsack k when the items $\mathcal{A}_k$ are already assigned. It is adapted from the notion of the value density from (Hiley, Julstrom, 2006).

When a full array of assignements is used as input, a full array of value densities is returned as

$$\mathrm{vd}(\mathcal{A}) = \begin{pmatrix} \mathrm{vd}_1(\mathcal{A}_1) & \mathrm{vd}_1(\mathcal{A}_2) & \cdots & \mathrm{vd}_1(\mathcal{A}_K) \\ \mathrm{vd}_2(\mathcal{A}_1) & \mathrm{vd}_2(\mathcal{A}_2) & \cdots & \mathrm{vd}_2(\mathcal{A}_K) \\ \vdots & \cdots & \ddots & \vdots \\ \mathrm{vd}_N(\mathcal{A}_1) & \mathrm{vd}_N(\mathcal{A}_2) & \cdots & \mathrm{vd}_N(\mathcal{A}_K) \end{pmatrix}$$

When the parameter reduced_output is set to True only a subset with the values of the unassigned items is returned.

Parameters

- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}.
- **weights** (list of float) – List of weights w_i of the N items that can be assigned.
- **assignments** (np.array or list of int) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. Alternatively, one can provide a list of the indices of selected items. In this case, it is assumed that these items are assigned to all knapsacks.
- **reduced_output** (bool, *optional*) – If set to True only the value density values of the unassigned objects are returned. Additionally, the indices of the unassigned items are returned as a second output.

Returns

- **densities** (`np.array`) – Array that contains the value densities of the objects. The length is equal to N, if `reduced_output` is `False`. If `reduced_output` is `True`, the return has length `len(densities)==len(unassigned_items)`. The number of dimensions is equal to the number of dimensions of `assignments`. Each column corresponds to a knapsack. If only a flat array is used as input, a flat array is returned.
- **unassigned_items** (`list`) – List of the indices of the unassigned items. This is only returned when `reduced_output` is set to `True`.

6.2 Module contents

class qmkpy.**QMKProblem**(*profits: Union[array, Iterable[Iterable]]*, *weights: Iterable[float]*, *capacities: Iterable[float]*, *algorithm: Optional[Callable] = None*, *args: Optional[tuple] = None*, *assignments: Optional[array] = None*, *name: Optional[str] = None*)

Bases: `object`

Base class to represent a quadratic multiple knapsack problem.

This class defines a standard QMKP with N items and K knapsacks.

profits

Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.

Type
`np.array`

weights

List of weights w_i of the N items that can be assigned.

Type
`list` of `float`

capacities

Capacities of the knapsacks. The number of knapsacks K is determined as `K=len(capacities)`.

Type
`list` of `float`

algorithm

Function that is used to solve the QMKP. It needs to follow the argument order `algorithm(profits, weights, capacities, ...)`.

Type
`Callable`, *optional*

args

Optional tuple of additional arguments that are passed to *algorithm*.

Type
`tuple`, *optional*

assignments

Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. This attribute is overwritten when calling *solve()*.

Type
np.array, *optional*

name

Optional name of the problem instance

Type
str, *optional*

classmethod load(*fname: str*, *strategy: str = 'numpy'*)

Load a QMKProblem instance

This functions allows loading a previously saved QMKProblem instance. The *save()* method provides a way of saving a problem.

See also:

save()
Method to save a QMKProblem instance which can then be loaded.

Parameters

- **fname** (str) – Filepath of the saved model
- **strategy** (str) – Strategy that is used to store the model. Valid choices are (case-insensitive):
  - numpy: Save the individual arrays of the model using the np.savez_compressed() function.
  - pickle: Save the whole object using Pythons pickle module
  - txt: Save the arrays of the model using the text-based format established by Billionnet and Soutif.
  - json: Save the arrays of the model using the JSON format.

Returns
problem – Loaded problem instance

Return type
QMKProblem

save(*fname: Union[str, bytes, PathLike]*, *strategy: str = 'numpy'*) → NoReturn

Save the QMKP instance

Save the profits, weights, and capacities of the problem. There exist different strategies that are explained in the strategy parameter.

See also:

load()
For loading a saved model.

Parameters

- **fname** (str or PathLike) – Filepath of the model to be saved at
- **strategy** (str) – Strategy that is used to store the model. Valid choices are (case-insensitive):

- `numpy`: Save the individual arrays of the model using the `np.savez_compressed()` function. See also *qmkpy.io.save_problem_numpy()*.
- `pickle`: Save the whole object using Pythons `pickle` module. See also *qmkpy.io.save_problem_pickle()*.
- `txt`: Save the arrays of the model using the text-based format established by Billionnet and Soutif. See also *qmkpy.io.save_problem_txt()*.
- `json`: Save the arrays of the model using the JSON format.

Return type
`None`

solve(*algorithm: Optional[Callable] = None*, *args: Optional[tuple] = None*) → Tuple[array, float]

Solve the QMKP

Solve the QMKP using `algorithm`. This function both returns the optimal assignment and the total resulting profit. This method also automatically sets the solution to the object's attribute *assignments*.

Parameters

- **algorithm** (`Callable`, *optional*) – Function that is used to solve the QMKP. It needs to follow the argument order `algorithm(profits, weights, capacities, ...)`. If it is `None`, the object attribute *algorithm* is used.
- **args** (`tuple`, *optional*) – Optional tuple of additional arguments that are passed to `algorithm`. If it is `None`, the object attribute *args* is used.

Returns

- **assignments** (`np.array`) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.
- **total_profit** (`float`) – Final total profit for the found solution.

qmkpy.**assignment_from_chromosome**(*chromosome: Iterable[int]*, *num_ks: int*) → array

Return the assignment matrix from a chromosome

Return the binary assignment matrix that corresponds to the chromosome. For more details about the connection between assignment matrix and chromosome check *chromosome_from_assignment()*.

See also:

chromosome_from_assignment()
For more details on the connection between assignment matrix and chromosome.

Parameters

- **chromosome** (`np.array` or `list` of `int`) – Chromosome version of `assignments`, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.
- **num_ks** (`int`) – Number of knapsacks K.

Returns
assignments – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Return type
`np.array`

qmkpy.**chromosome_from_assignment**(*assignments: array*) → Iterable[int]

Return the chromosome from an assignment matrix

The chromosome version of `assignments` is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Example

Assume that we have 4 items and 3 knapsacks. Let Items 1 and 4 be assigned to Knapsack 1, Item 2 is assigned to Knapsack 3 and Item 3 is not assigned. In the binary representation, this is

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

The corresponding chromosome is

$$C(A) = \begin{pmatrix} 1 & 3 & 0 & 1 \end{pmatrix}$$

However, in the 0-index based representation in Python, this function will return

```
chromosome_from_assignment(A) = [0, 2, -1, 0]
```

as the chromosome.

Parameters

assignments (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Returns

chromosome – Chromosome version of `assignments`, which is a list of length N where $c_i = k$ means that item i is assigned to knapsack k. If the item is not assigned, we set $c_i = -1$.

Return type

np.array

qmkpy.**total_profit_qmkp**(*profits: array*, *assignments: array*) → float

Calculate the total profit for given assignments.

This function calculates the total profit of a QMKP for a given profit matrix P and assignments $\mathcal{A}$ as

$$\sum_{u=1}^{K} \left(\sum_{i \in \mathcal{A}_u} p_i + \sum_{\substack{j \in \mathcal{A}_u \\ j \neq i}} p_{ij} \right)$$

where $\mathcal{A}_u$ is the set of items that are assigned to knapsack u.

Parameters

- **profits** (np.array) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}. The profit of the single items p_i corresponds to the main diagonal elements, i.e., $p_i = p_{ii}$.
- **assignments** (np.array) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j.

Returns

Value of the total profit

Return type
`float`

qmkpy.**value_density**(*profits: array*, *weights: Iterable[float]*, *assignments: Union[array, Iterable[int]]*, *reduced_output: bool = False*) → Iterable[float]

Calculate the value density given a set of selected objects.

This function calculates the value density of item i for knapsack k and given assignments $\mathcal{A}_k$ according to

$$\mathrm{vd}_i(\mathcal{A}_k) = \frac{1}{w_i}\left(p_i + \sum_{\substack{j\in\mathcal{A}_k \\ j\neq i}} p_{ij}\right)$$

This value indicates the profit (per item weight) that is gained by adding the item i to knapsack k when the items $\mathcal{A}_k$ are already assigned. It is adapted from the notion of the value density from (Hiley, Julstrom, 2006).

When a full array of assignements is used as input, a full array of value densities is returned as

$$\mathrm{vd}(\mathcal{A}) = \begin{pmatrix} \mathrm{vd}_1(\mathcal{A}_1) & \mathrm{vd}_1(\mathcal{A}_2) & \cdots & \mathrm{vd}_1(\mathcal{A}_K) \\ \mathrm{vd}_2(\mathcal{A}_1) & \mathrm{vd}_2(\mathcal{A}_2) & \cdots & \mathrm{vd}_2(\mathcal{A}_K) \\ \vdots & \cdots & \ddots & \vdots \\ \mathrm{vd}_N(\mathcal{A}_1) & \mathrm{vd}_N(\mathcal{A}_2) & \cdots & \mathrm{vd}_N(\mathcal{A}_K) \end{pmatrix}$$

When the parameter `reduced_output` is set to `True` only a subset with the values of the unassigned items is returned.

Parameters

- **profits** (`np.array`) – Symmetric matrix of size $N \times N$ that contains the (joint) profit values p_{ij}.
- **weights** (`list` of `float`) – List of weights w_i of the N items that can be assigned.
- **assignments** (`np.array` or `list` of `int`) – Binary matrix of size $N \times K$ which represents the final assignments of items to knapsacks. If $a_{ij} = 1$, element i is assigned to knapsack j. Alternatively, one can provide a list of the indices of selected items. In this case, it is assumed that these items are assigned to all knapsacks.
- **reduced_output** (`bool`, *optional*) – If set to `True` only the value density values of the unassigned objects are returned. Additionally, the indices of the unassigned items are returned as a second output.

Returns

- **densities** (`np.array`) – Array that contains the value densities of the objects. The length is equal to N, if `reduced_output` is `False`. If `reduced_output` is `True`, the return has length `len(densities)==len(unassigned_items)`. The number of dimensions is equal to the number of dimensions of `assignments`. Each column corresponds to a knapsack. If only a flat array is used as input, a flat array is returned.
- **unassigned_items** (`list`) – List of the indices of the unassigned items. This is only returned when `reduced_output` is set to `True`.

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[AGH22] Méziane Aïder, Oussama Gacem, and Mhand Hifi. "Branch and solve strategies-based algorithm for the quadratic multiple knapsack problem", Journal of the Operational Research Society, vol. 73, no. 3, pp. 540-557. (2022) DOI: 10.1080/01605682.2020.1843982

[HJ06] Amanda Hiley and Bryant A. Julstrom. "The quadratic multiple knapsack problem and three heuristic approaches to it", Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO '06), pp. 547–552. (2006) DOI: 10.1145/1143997.1144096

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. "Knapsack Problems", Springer Berlin Heidelberg. (2004) DOI: 10.1007/978-3-540-24777-7

PYTHON MODULE INDEX

q

INDEX

S

T

V

W