SIGEVO Summer School (S3)

Modelling Optimization Problems for Evolutionary Algorithms and Other Metaheuristics

Project statement

Carlos M. Fonseca and Alexandre B. Jesus
Department of Informatics Engineering
University of Coimbra
Portugal

13 July 2023

The projects consist in modelling given combinatorial optimization problems both as local-search and constructive-search problems and implementing the corresponding models bearing in mind the principles and ideas presented in the lecture. Problem implementations shall adhere to an Application Programming Interface (API) so that a number of simple algorithms (provided together with the API specification) can run directly on them in a problem-independent way, and experimental results can be collected.

Six different problems are proposed. Each problem should be modelled and implemented in plain Python (version 3) by a group of 4 to 6 students (typically 5). It is essential that most, but not necessarily all, members of each group are comfortable programming in Python, so that coding work can be effectively shared during the time available. Group members who do not program should still be able to contribute actively to other aspects of the project work, including conceptual modelling, experimentation, and analysis and discussion of results.

It is expected that each group gives a presentation in the last session of the School.

Presentations should cover:

1. A description of the problem considered
2. Modelling decisions, such as the definition of the decision space, construction rule, and neighbourhood structure
3. Implementation decisions, including solution, component and local move representation, generation, and evaluation
4. Preliminary experimental results

Software requirements

A text editor and a recent Python3 installation is all that is required to carry out the proposed projects. Ideally, the PyPy (www.pypy.org) alternative implementation of Python should be used if possible (the latest version is v7.3.12), but the standard Python interpreter (v3.9 or above) will also work. Older versions may work, too.

Students should ensure that they have a working Python programming environment before the School starts.

Programming skills

Project code will consist of a set of classes with pre-defined methods to be implemented by the students. Template source files and problem instances will be provided for each project.

1 Problem descriptions

1.1 3D printing service

An additive manufacturing startup provides 3D printing services to both private and business customers. A single 3D printer is used to produce custom metal parts that must be printed without interruption. Once a part is printed and is allowed to cool down, it is ready for delivery and the machine becomes available to print the next part. Each part must be ready by a given deadline, which is set at the time the order is placed. Missing the deadline leads to a penalty that depends on the part, and is proportional to how late the part is completed. The time needed to print a given part, including the cooldown time, can be determined in advance.

The problem consists in determining the order in which a set of parts should be manufactured on the single metal 3D printer so as to minimize the total penalty

incurred by the service provider.

1.1.1 Input

The input contains $3N$ integers separated by whitespace (spaces, tabs, or newlines). The first N integers are the processing times p_j, the next N integers are the weights w_j, and the last N integers are the due dates d_j of the various parts $j = 1, \ldots, N$.

1.1.2 Solution evaluation

A feasible solution is a permutation $\pi = (j_1, j_2, \ldots, j_N)$ of the integers $1, \ldots, N$, where j_i denotes the index of the i-th part to be printed. The cost of a solution π is the total weighted tardiness:

$$\sum_{j=1}^{N} w_j T_j$$

where $T_j = \max\{C_j - d_j, 0\}$ denotes the tardiness of job j, and C_j denotes the completion time of job j. Given the permutation π, the completion time of job j_i is

$$C_{j_i} = \sum_{k=1}^{i} p_{j_k}$$

for each $i = 1, \ldots, N$.

1.1.3 Output

Print the part indices $j_1, j_2, \ldots, j_N$ of the N parts in the order in which they should be processed separated by whitespace.

1.1.4 Example

Input

```
26    24    79    46    32
1     10     9    10    10
80   220   180    50   100
```

Output (cost 67)

```
4 1 5 3 2
```

1.2 Urban waste collection

Waste collection vehicles are used to regularly empty the many public waste containers located on the side of roads (away from intersections) across Neat City. A given city quarter is served by a single vehicle that is large enough to collect the waste from all containers in a single tour. A collection tour consists in the vehicle leaving the depot, visiting and emptying all containers in the quarter, disposing of the collected waste at the city's waste treatment plant, and returning to the depot.

On one-way and most two-way roads, the vehicle may empty the containers it passes by regardless of whether or not it is driving on the side of the road where the container is located. On multi-lane roads, however, the vehicle and the container must be on the same side of the road for the collection to be possible. In any case, no U-turns are allowed at waste containers. In other words, denoting the two possible directions by 0 and 1, a vehicle arriving to a container in one direction must depart from that container in the same direction.

Suppose that the cost of a waste collection tour is proportional to the total distance travelled by the vehicle until it returns to the depot. The problem consists in determining the shortest tour subject to the above one-way, roadside, and no U-turn constraints.

1.2.1 Input

The first line of the input contains a single integer, N, denoting the number of containers to visit. The rest of the input consists of $4+4N$ additional lines, each containing N integers separated by whitespace (spaces or tabs).

The first of those lines contains the distances from the depot to each container, arriving to the container in direction 0. The next line contains the distances from the depot to each container, but arriving to the container in direction 1. Similarly, the following line contains the distances from each container to the treatment plant, departing in direction 0, and the same for the next line, when departing in direction 1.

The next 4 sequences of N lines are distance matrices for the N containers, for each combination of directions. The value at position $(r,c) \in \{1,\ldots,N\}^2$ of the first matrix is the distance from container r to container c when departing from r in direction 0 and arriving at c also in direction 0. The other three matrices are analogous, corresponding to directions 0–1, 1–1 and 1–0, respectively. If a given direction is not possible for a container, the applicable rows or columns of the corresponding matrices take the value -1.

1.2.2 Solution evaluation

The cost of a solution is the total distance travelled from the depot to the first container, then to the second, and so on, and finally from the last container to the treatment plant, taking into account the directions at the containers. The distance between the treatment plant to the depot is a constant and can be ignored for the purpose of optimization.

1.2.3 Output

Print on consecutive lines the indices of the N containers, and the corresponding directions, in the order in which they should be visited. Each line should contain a container index, in $\{1,\ldots,N\}$, and the corresponding direction number separated by whitespace.

1.2.4 Example

Input

```
3
133 21 93
52 171 29
184 88 57
56 121 148
0 156 228
112 0 72
49 129 0
188 245 164
32 150 8
1 179 37
0 178 36
65 0 42
137 142 0
253 141 213
113 147 64
217 105 177
```

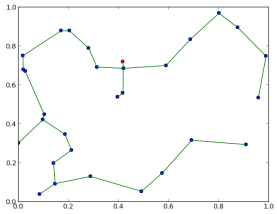
Output (cost 490)

```
3 1
1 0
2 0
```

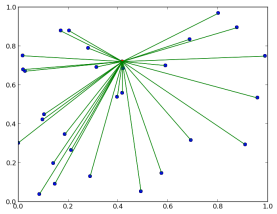
1.3 Campus network

New network infrastructure is to be installed at a University campus. Each building on campus will be connected directly to the computing centre by a dedicated high-speed cable. Trenches will be dug in order to allow cables to be laid from one building to another. Once dug, a trench may be used to lay more than one cable, such as those connecting other buildings farther away from the computing centre. Note that trenches may only be dug from one building to another, and that different trenches may not cross each other.

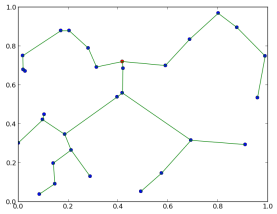
The cost of digging a trench is proportional to its length, as is the cost of a cable. The problem consists in determining what trenches to dig so as to minimize the total (trench and cable) set up cost of the new network.



Trench-optimal solution



Cable-optimal solution



Optimal solution

1.3.1 Input

The first line of the input contains an integer, N, denoting the number of buildings, including the computing centre itself, followed by two real values denoting the unit costs of the cable and of digging trenches. The three values are separated by whitespace.

The following N lines contain the x and y coordinates of the buildings separated by whitespace, one building per line. The first building is the computing centre.

1.3.2 Solution evaluation

The cost of a solution is the total cost of the cable and trenches. The cost of the cable is its unit cost times the sum of the lengths of the paths through the trenches from each building to the computing centre. The cost of the trenches is their unit cost times their total length.

1.3.3 Output

For each pair of buildings connected by a trench, print the indices of the two buildings separated by whitespace on a different line. The computing centre has index 1.

1.3.4 Example

Input

```
6 0.2 1.8
985.087 238.729
623.627 292.121
310.813 373.334
95.470 264.070
712.352 759.543
104.789 562.537
```

Output (cost 3792.645)

```
1 2
2 3
2 5
3 4
3 6
```

1.4 Community detection

A fully-connected undirected graph, where vertices represent users and weighted edges represent the intensity of some attribute of their interaction that may be positive or negative, was obtained from social network data. Users connected by edges with positive weight show affinity to each other, whereas negative edge weights indicate lack of affinity. Groups of users connected mostly by positively weighted edges suggest the existence of a community involving those users.

The problem consists in finding the groups of users that maximize the total internal edge weight of all groups.

1.4.1 Input

The first line of the input contains an integer N denoting the number of users.

The following N lines give the weights between every pair of users. This is given as an upper triangular matrix. In particular, the first line contains the weight between user 1 and all users from 1 to N. The second line contains the weights between user 2 and all users from 2 to N. And so on.

Weights are symmetric, i.e., $w_{i,j} = w_{j,i}$.

1.4.2 Solution evaluation

A solution is a set of M sets $S = \{S_1, S_2, \ldots, S_M\}$, such that each set S_i, $1 \leq i \leq M$, denotes a community. For the solution to be feasible, the intersection of every pair of distinct communities must be empty, i.e., $S_i \cap S_j = \emptyset$, $i \neq j$, and the union of all communities must be the set of all numbers from 1 to N, i.e., $\bigcup_{i=1}^{M} S_i = \{1, 2, \ldots, N\}$.

The objective value of a solution S is given by the sum of all connections within each community, i.e.:

$$\sum_{i=1}^{M} \frac{\sum_{j \in S_i} \sum_{k \in S_i} w_{j,k}}{2}$$

1.4.3 Output

Print one community per line. For each community, print a list of space-separated integers denoting the users of that community in a separate line.

1.4.4 Example

Input

```
4
0 1 -2 3
0 2 -1
0 3
0
```

Output (score 3)

```
1 2 4
3
```

1.5 Laptop assembly

1.5.1 Description

A laptop manufacturer pre-assembles different laptop models of several base designs in order to better meet customer demand. For a given base design, there is a number, M, of different laptop models. Assembling one laptop unit of a given model requires certain numbers of parts of up to P types. Laptops are assembled in mixed-model manufacturing cells. At the beginning of each day, the number of units, d_m, of each model m, $0 \leq m < M$, to be assembled at a given cell is defined. Thus, the total number of units to be assembled at that cell is $T = \sum_{m=0}^{M-1} d_m$. Laptop units are assembled in sequence, requiring one time slot each. Since T units are to be assembled in one day, there are also T time slots.

In order to ensure a smooth flow of parts to the manufacturing cell and reduce part safety stock levels, it is desired that the rate of consumption of the different parts throughout the day is kept as close to constant as possible. In other words, the number of parts of type p, $0 \leq p < P$, used up to slot t, $1 \leq t \leq T$, should be close to $t \cdot r_p$, where r_p is the total number of parts of type p required to build all d_m units of all models m, $0 \leq m < M$, divided by T.

The problem consists in determining the order in which the various units of the different models should be assembled to achieve such a smooth flow of parts.

1.5.2 Input

The first line of the input contains two integers, M and P separated by whitespace, where M is the number of different models to be assembled, and P is the number of different types of parts to be used.

The second line contains M space-separated integers, $d_1, d_2, \ldots, d_M$, corresponding to the number of units of each model to assemble.

The following P lines give a space-separated matrix of the number of parts of each type that are used in each model:

$$\begin{array}{ccccc} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,M-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,M-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{P-1,0} & a_{P-1,1} & a_{P-1,2} & \cdots & a_{P-1,M-1} \end{array}$$

where $a_{p,m}$, $0 \leq p < P$ and $0 \leq m < M$, denotes the number of parts of type p needed to assemble one model m unit.

1.5.3 Solution evaluation

A feasible solution is a vector $u = (u_1, u_2, \ldots, u_T)$ such that $0 \leq u_t < M$ and that $\sum_{t=1}^{T} [u_t = m] = d_m$ for all $0 \leq m < M$, where $[u_t = m]$ denotes the value 1 if $u_t = m$ and 0 otherwise.

The cost of a solution u is the sum of the squared deviations between the target demand and cumulated actual demand for all part types and time slots. Mathematically:

$$J(u) = \sum_{t=1}^{T} \sum_{p=0}^{P-1} \left(t \cdot r_p - \sum_{i=1}^{t} a_{p,u_i} \right)^2$$

1.5.4 Output

Print the model indices $u_1, u_2, \ldots, u_T$ of the T units to be assembled, one value per line.

1.5.5 Example

Input

```
4 5
2 2 2 4
1 1 0 0
0 0 0 1
1 0 0 0
2 1 0 0
1 1 1 0
```

Output ($J(u) = 9.6000$)

```
2
1
3
0
3
2
1
3
0
3
```

1.6 Candle Race

1.6.1 Description

In a race, there is a number of villages to be visited, N, and there is a candle in each village. Villages and the corresponding candles are identified by an index, $i = 1, \ldots, N$, and different candles may have different properties. In particular, each candle i has a length, h_i, and a burning rate, b_i. For each minute that passes, the length h_i is reduced by b_i, until it eventually reaches 0. There is also a village with index 0 where the race starts. This village has no candle.

At the beginning of the race, the candles are lit. The goal is to visit each village and blow out the candles. When a candle is blown out, the player scores points equal to the remaining length of the candle. Given the properties of each candle, and the minimum travelling time between the villages, the problem consists in finding a route that maximizes the player's score.

1.6.2 Input

The first line of the input contains an integer N denoting the number of villages.

The following line contains two integers denoting the x and y coordinates of the starting village.

The following $N-1$ lines each give four space-separated integers corresponding to the information pertaining to each village:

- x coordinate
- y coordinate
- Initial candle height
- Burning rate per unit of time

The time between two village is given by the Manhattan distance.

1.6.3 Output

Print the indices of the villages in the order that they should be visited (excluding the starting village), one index per line. It is assumed that, once the player arrives at a village, they immediately blow out the candle (there is no point in waiting after all).

Note: The player does **not** need to visit every village.

1.6.4 Example

Input

```
5
0 0
16 25 464 2
10 34 696 6
28 17 302 5
19 57 523 10
```

Output (Score 568)

```
1
3
2
4
```

2 Problem modelling

Modelling each of the above problems as a search problem begins with attempting to answer the following questions:

Problem instance What (known) data is required to fully characterize a particular *instance* of the problem? This must be available in advance, and is not changed by the solver in any way.

Solution What (unknown) data is required to fully characterize a (feasible) candidate *solution* to a given problem instance? This is the data needed to implement the solution in practice, and will be determined by the solver during the optimization run.

Objective function How can the performance of a given candidate solution be measured? This depends only on the problem instance and the actual solution itself, and never on how the solution was actually found. Is the corresponding value to be minimized or maximized?

Combinatorial structure (Constructive search) How can a solution be constructed piece by piece? If solutions are seen as *subsets* of a larger *ground set* of solution components, the construction process consists simply in successively adding components to the empty set according to some *construction rule*. The *partial solutions* generated during the construction process represent *all feasible solutions* that contain them, and their performance can be

inferred from those sets in terms of suitable *lower bounds* (minimization) or *upper bounds* (maximization).

Neighbourhood structure (Local search) What makes two given solutions *similar* to each other? This usually means that parts of the two solutions are somehow identical, but it should also happen that they exhibit *similar* performance *in most cases*. A candidate solution that performs at least as well as all solutions similar to it (its *neighbours*) is called a *local optimum* of the corresponding problem instance.

The choice of the neighbourhood structure is particularly important for the success of local-search algorithms. By ensuring that similar solutions tend to exhibit similar performance, one seeks to induce fewer local optima and large basins of attraction to those optima, although this can seldom be guaranteed. Furthermore, any two feasible solutions should be connected by a sequence of consecutive neighbours, so that the unknown global optimum can, at least in principle, be reached from any initial solution.

Similarly, the choice of ground set and construction rule, and the quality of the associated bounds, are very important for the success of constructive-search algorithms. In particular, it should be possible to construct all feasible solutions, and the inferred quality of partial solutions should be as accurate as possible so that the construction process can effectively rely on it.

2.1 Computational model

Once suitable answers to the above questions are obtained, a more refined set of questions relative to the computer implementation of the model can be considered.

Problem instance representation How should the problem instance data be stored in a data structure so that the objective function and corresponding bounds can be easily computed?

Solution representation How should (possibly incomplete) solutions be represented, i.e., stored as a data structure, so that:

1. Their performance can be evaluated efficiently through the objective function or related bounds?
2. **[Constructive search]** Feasible solutions can be easily constructed by successively adding components?
3. **[Local search]** Solutions can be easily modified to obtain neighbouring solutions?

Solution evaluation How can the objective function and/or corresponding bounds be computed given the instance data and the solution representation?

Move representation How can *moves* be represented, i.e.,

[Constructive search] the *addition* or *removal* of components to/from a (partial) solution?

[Local search] changes that, when applied to a solution, lead to a neighbouring solution?

Solution modification What are *valid* moves, and how are they applied to a solution?

Incremental solution evaluation When an evaluated (partial) solution is modified by applying one or more moves to it, can the resulting solution be evaluated faster than would otherwise be the case? How?

Move evaluation How much would applying a given move to a (partial) solution change its performance? Can this effect be computed more efficiently without modifying the original solution than by evaluating it, applying the move and evaluating the result? How?

3 Application Programming Interface (API)

Several algorithms, implemented in `Python3`, are provided in the folder `api`, along with some utility functions. These algorithms assume a common API for the definition of the computational model and operators described above. In this section we start by showing all methods that need to be implemented to run **all** provided algorithms. In Section 3.5 we will describe the methods that are required by each algorithm and what parameters they may take.

The code provided is also available in the file `base.py` along with a "main" block that allows choosing the algorithms to run, time budgets, input, and output files, and logging information. Note that we use type annotations to make the parameters and return types clearer. As a result, the library requires an up to date installation of `Python` version 3.7 or later.

3.1 Component class

The `Component` class defines a component for a solution. It can have any number of fields. There is only one reserved property, `cid` (component id), which is required for ant colony optimization algorithms (`aco` and `mmas`).

```
class Component:
    @property
    def cid(self) -> Hashable:
        raise NotImplementedError
```

3.2 LocalMove class

The `LocalMove` class defines a local move for a solution. It can have any number of fields and no methods or properties are currently required by the API.

```
class LocalMove:
    ...
```

3.3 Solution class

The `Solution` class defines the solution itself and may implement several methods which are described below. Which methods to implement depends on the algorithms that we want to use.

One outlier is the `output` method, which is not needed by any algorithm. Instead, it is used by the "main" block of `base.py` to print the final solution.

Finally, note that the `Component` and `LocalMove` type refers to the two classes describe previously. Moreover, the `Objective` type refers to the type of the objective value. In `base.py` we set this to `Any` type since it depends on the particular problem and model. If you want to take advantage of type checking you can change this to the appropriate type for your problem.

```
class Solution:
    def output(self) -> str:
        """
        Generate the output string for this solution
        """
        raise NotImplementedError

    def copy(self) -> Solution:
        """
        Return a copy of this solution.

        Note: changes to the copy must not affect the original
        solution. However, this does not need to be a deepcopy.
        """
        raise NotImplementedError
```

```
def is_feasible(self) -> bool:
    """
    Return whether the solution is feasible or not
    """
    raise NotImplementedError

def objective(self) -> Optional[Objective]:
    """
    Return the objective value for this solution if defined, otherwise
    should return None
    """
    raise NotImplementedError

def lower_bound(self) -> Optional[Objective]:
    """
    Return the lower bound value for this solution if defined,
    otherwise return None
    """
    raise NotImplementedError

def add_moves(self) -> Iterable[Component]:
    """
    Return an iterable (generator, iterator, or iterable object)
    over all components that can be added to the solution
    """
    raise NotImplementedError

def local_moves(self) -> Iterable[LocalMove]:
    """
    Return an iterable (generator, iterator, or iterable object)
    over all local moves that can be applied to the solution
    """
    raise NotImplementedError

def random_local_move(self) -> Optional[LocalMove]:
    """
    Return a random local move that can be applied to the solution.

    Note: repeated calls to this method may return the same
    local move.
```

```
        """
        raise NotImplementedError

    def random_local_moves_wor(self) -> Iterable[LocalMove]:
        """
        Return an iterable (generator, iterator, or iterable object)
        over all local moves (in random order) that can be applied to
        the solution.
        """
        raise NotImplementedError

    def heuristic_add_move(self) -> Optional[Component]:
        """
        Return the next component to be added based on some heuristic
        rule.
        """
        raise NotImplementedError

    def add(self, component: Component) -> None:
        """
        Add a component to the solution.

        Note: this invalidates any previously generated components and
        local moves.
        """
        raise NotImplementedError

    def step(self, lmove: LocalMove) -> None:
        """
        Apply a local move to the solution.

        Note: this invalidates any previously generated components and
        local moves.
        """
        raise NotImplementedError

    def objective_incr_local(self, lmove: LocalMove) -> Optional[Objective]:
        """
        Return the objective value increment resulting from applying a
        local move. If the objective value is not defined after
        applying the local move return None.
```

```
        """
        raise NotImplementedError

    def lower_bound_incr_add(self, component: Component) -> Optional[Objective]:
        """
        Return the lower bound increment resulting from adding a
        component. If the lower bound is not defined after adding the
        component return None.
        """
        raise NotImplementedError

    def perturb(self, ks: int) -> None:
        """
        Perturb the solution in place. The amount of perturbation is
        controlled by the parameter ks (kick strength)
        """
        raise NotImplementedError

    def components(self) -> Iterable[Component]:
        """
        Returns an iterable to the components of a solution
        """
        raise NotImplementedError
```

3.4 Problem class

Finally, the `Problem` class defines a problem. It needs to implement the two methods described below for the "main" block provided in `base.py`.

```
class Problem:
    @classmethod
    def from_textio(cls, f: TextIO) -> Problem:
        """
        Create a problem from a text I/O source `f`
        """
        raise NotImplementedError

    def empty_solution(self) -> Solution:
        """
        Create an empty solution (i.e. with no components).
        """
```

```
        raise NotImplementedError
```

3.5 Algorithm requirements

In this section we briefly explain the call signatures of the algorithms, and describe the methods that need to be implemented.

One aspect to note is that all algorithms are designed to receive and return a solution. Moreover, the solution that is passed as a parameter may be altered. As such, if you want to keep the original solution please make a copy before calling the algorithm.

3.5.1 Heuristic construction (`heuristic_construction`)

Heuristic construction has the following signature:

```
def heuristic_construction(solution: Solution) -> Solution
```

The `Solution` class needs to implement the following methods:

- `add`
- `heuristic_add_move`

3.5.2 Greedy construction (`greedy_construction`)

Greedy construction has the following signature.

```
def greedy_construction(solution: Solution) -> Solution
```

The `Solution` class needs to implement the following methods:

- `add`
- `add_moves`
- `lower_bound_incr_add`

Furthermore, the `Objective` type should be order comparable (i.e., it should implement the `__lt__` method).

3.5.3 Beam search (`beam_search`)

Beam search has the following signature:

```
def beam_search(solution: Solution, bw: int = 10) -> Solution
```

where `bw` denotes the beam width. The `Solution` class needs to implement the following methods:

- `lower_bound`
- `objective`
- `copy`
- `is_feasible`
- `add_moves`
- `lower_bound_incr_add`
- `add`

Moreover, the `Objective` type should be order comparable (i.e., it should implement the `__lt__` method), and be addable (i.e., it should implement the `__add__` method).

3.5.4 GRASP (`grasp`)

GRASP has the following signature:

```
def grasp(solution: Solution,
          budget: float,
          alpha: float = 0.1,
          seed: Optional[int] = None,
          local_search: Optional[LocalSearch[Solution]] = None,
          ) -> Optional[Solution]
```

where `budget` is the time budget for the algorithm, `alpha` is a parameter to control the range of components that should be considered for selection according to their quality as given by the increment in terms of lower bound, `seed` is the seed for the random selection, and `local_search` is an optional local search method that receives and returns a solution.

The `Solution` class needs to implement the following methods:

- `lower_bound`
- `objective`
- `copy`
- `is_feasible`
- `add_moves`

- `lower_bound_incr_add`
- `add`

Moreover, the `Objective` type should be a real or integer number.

3.5.5 Ant system (`ant_system`)

Ant system is a basic ant system with following signature:

```
def aco(solutions: list[Solution],
        budget: float,
        tau0: float,
        alpha: float = 1.0,
        beta: float = 3.0,
        rho: float = 0.5,
        seed: Optional[int] = None,
        local_search: Optional[LocalSearch[Solution]] = None,
        ) -> Optional[Solution]
```

where `budget` is the time budget for the algorithm, `tau0`, `alpha`, `beta`, `rho` are the parameters of the algorithm, `seed` is the seed for the random selection, and `local_search` is an optional local search method that receives and returns a solution.

The `Solution` class needs to implement the following methods:

- `lower_bound`
- `objective`
- `copy`
- `is_feasible`
- `add_moves`
- `lower_bound_incr_add`
- `add`
- `components`

Moreover, the `Objective` type should be a real or integer number.

3.5.6 Max-min ant system (`mmas`)

MMAS has the following signature:

```
def mmas(solutions: list[Solution],
         budget: float,
         taumax: float,
         a: float = 5.0,
         alpha: float = 1.0,
         beta: float = 3.0,
         rho: float = 0.5,
         globalratio: float = 0.5,
         nrestart: int = 500,
         seed: Optional[int] = None,
         local_search: Optional[LocalSearch[Solution]] = None,
         ) -> Optional[Solution]
```

where most parameters are the same as `aco`. The parameter `taumax` defines the maximum value for the pheromones, `a` is used to compute `taumin`, in particular `taumin = a * taumax`, `globalratio` gives the probability of using the global best to update the pheromones, as opposed to the iteration best, and `nrestart` gives the number of iterations without improvements that are needed for a restart occur.

The `Solution` class needs to implement the following methods:

- `lower_bound`
- `objective`
- `copy`
- `is_feasible`
- `add_moves`
- `lower_bound_incr_add`
- `add`
- `components`

Moreover, the `Objective` type should be a real or integer number.

3.5.7 Best improvement (`best_improvement`)

Best improvement has the following signature:

```
def best_improvement(solution: Solution, budget: float) -> Solution
```

where `budget` gives the time budget for the local search method.

The `Solution` class needs to implement the following methods:

- `local_moves`
- `objective_incr_local`
- `step`

Moreover, the `Objective` type should be a real or integer number.

3.5.8 First improvement (`first_improvement`)

First improvement has the following signature:

```
def first_improvement(solution: Solution, budget: float) -> Solution
```

where `budget` gives the time budget for the local search method.

The `Solution` class needs to implement the following methods:

- `random_local_moves_wor`
- `objective_incr_local`
- `step`

Moreover, the `Objective` type should be a real or integer number.

3.5.9 Random local search (`rls`)

Random local search has the following signature:

```
def rls(solution: Solution, budget: float) -> Solution
```

where `budget` gives the time budget for the local search method.

The `Solution` class needs to implement the following methods:

- `random_local_moves_wor`
- `objective_incr_local`
- `step`

Moreover, the `Objective` type should be a real or integer number.

3.5.10 Iterated local search (`ils`)

ILS has the following signature:

```
def ils(solution: Solution, budget: float, ks: int = 3) -> Solution
```

where budget gives the time budget for the local search method, and ks gives the kick strength for the perturb method.

The Solution class needs to implement the following methods:

- objective
- copy
- random_local_moves_wor
- objective_incr_local
- step
- perturb

Moreover, the Objective type should be a real or integer number.

3.5.11 Simulated annealing (sa)

SA has the following signature:

```
def sa(solution: Solution,
       budget: float,
       init_temp: float,
       seed: Optional[int] = None,
       temperature: Optional[Callable[[float], float]] = None,
       acceptance: Optional[Callable[[float, float], float]] = None,
       ) -> Solution
```

where budget gives the time budget for the local search method, init_temp gives the kick strength for the perturb method, temperature is a function that defines how temperature decays over time (if None is given we consider a linear decay function stopping at 0), and acceptance defines the acceptance function (if None is given we consider an exponential acceptance function).

The Solution class needs to implement the following methods:

- objective
- copy
- random_local_moves_wor
- objective_incr_local
- step

- `perturb`

Moreover the `Objective` type should be a real or integer number.