SIGEVO Summer School (S3)

Modelling Optimization Problems for Evolutionary Algorithms and Other Metaheuristics

Problem modelling and API documentation

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

13 July 2023

1 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.

1.1 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?

2 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 2.5 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.

2.1 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
```

2.2 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:
    ...
```

2.3 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
```

2.4 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
```

2.5 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.

2.5.1 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`

2.5.2 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).

2.5.3 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).

2.5.4 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.

2.5.5 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.

2.5.6 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.

2.5.7 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.

2.5.8 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.

2.5.9 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.

2.5.10 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.

2.5.11 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.

Acknowledgements

This work is partially funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit – UIDB/00326/2020 or project code UIDP/00326/2020.