ROAR-NET API Specification

About ROAR-NET API

The ROAR-NET API defines a **framework for black-box optimisation problem modelling**. It is based on abstractions aimed at unifying black-box optimisation problems, and on a clear **separation between problem modelling and randomised optimisation algorithms**. The resulting standardisation of black-box problem modelling leads to the following advantages:

- Different optimisation algorithms can operate on the same problem model, e.g., branch-and-bound and local-search.
- Different optimisation problems (or models) can be solved by the same algorithms.

Conceptualisation

The goal of this specification is to define a minimal, programming-language agnostic, interface to the optimisation problems that a wide variety of **Randomised Optimisation Algorithms (ROAs)** can use to solve these problems. In doing this, the specification should facilitate modelling of real life problems that can be solved by a wide range of algorithms and make it possible to easily benchmark problem models and ROAs.

The specification defines the **types** and **operations** that have to be implemented when modelling an optimisation problem in order to allow supported ROAs to function.

The API specification has the following design principles:

- it is **agnostic of the programming language**.
- it is **agnostic of the programming paradigm**.
- requires implementations of the API to **only store problem-specific information**. They must be completely free of any information about the optimisation algorithm to be used.

Supported features

The API supports a broad range of problems and Randomised Optimisation Algorithms (ROAs). The current version of the specification covers, but is not limited to, the following classes of **single-objective** optimisation problems:

- **combinatorial problems**
- **discrete problems**

Furthermore, it considers the following main algorithmic approaches:

- **Constructive search**: solutions are constructed iteratively. For example, by adding a component, or assigning the value of a decision variable, on each iteration.
    - Example of algorithms include, backtracking, branch-and-bound, greedy algorithms, and

GRASP.
    - Some of these algorithms may provide guarantees on the quality of the solution.
- **Local search**: solutions are found by modifying feasible solutions. For example, by exchanging some components of the solution, or changing the value of a decision variable.
    - Examples of algorithms include, first and best improvement local search, iterated local search, and simulated annealing.
    - These algorithms typically do not provide a guarantee on the quality of the solution found but are usually able to find good approximations in a reasonable amount of time.

Extensions to multiple objectives, uncertainty settings, and support for further algorithms are currently under discussion on the Github repository, where everyone is welcome to contribute and propose new ideas.

Acknowledgement

This specification is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

COST (European Cooperation in Science and Technology) is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career and innovation.

Background

Optimisation problem

Formally, we consider a **(minimisation) problem** in the form

$$\min\{f(x) \mid x \in F\}$$

The set F is the set of **feasible solutions**, which is a subset of the **decision space**, S. The mapping $f : S \to \mathbb{R}$ is an **objective function** that associates to each solution $s \in S$ a real value in the **objective space**, $\mathbb{R}$.

In this case, the objective space is totally ordered, and the minimum is given by all solutions $x^* \in F$ such that $f(x^*) \leq f(x)$ holds for all $x \in F$. Such a solution x^* is called an **optimal solution**. **Maximisation problems**, in which we are searching for the feasible solutions whose value is the maximum in the objective space, can be reduced to minimisation problems as $\max\{f(s) \mid s \in F\} = -\min\{-f(s) \mid s \in F\}$.

In this specification, optimisation problems are specified by means of a `Problem` type.

Search space

The representation and the exploration of the **search space** of a problem implementing the API rely on how solutions are characterised, how they are generated, and on how they are evaluated.

Candidate solutions

We assume that the details regarding the components of the solution are not exposed, nor the details of how to create or modify a solution. Instead, the API provides abstract operations that allow to generate and modify solutions in a black-box manner, and that is common to all algorithmic approaches covered.

The API relies on the observation that:

- constructive search algorithms work by starting from an initial *state* and sequentially applying *actions* or *changes* to it to attain new states until a goal is reached.
- local search algorithms work by maintaining a set of states, and iterating through states that can be reached by applying (small) *changes* to the stored states.

In both cases, such states represent **candidate solutions**. The changes that allow to modify a solution s to obtain another solution s' is called a **move**, in which case s' is called a **neighbour** of s.

A **candidate solution** is an element of the *decision space*, $s \in S$, but not necessarily an element of the set of feasible solutions. Such solution can violate constraints or satisfy other constraints that are known to be necessary for optimal solutions. A candidate solution in ROAs can be:

- a **partial solution**, which may not contain yet all components of a feasible solution.

- a **complete solution**, which contains all components but may violate problem constraints and may not be optimal.

In this specification, solutions are specified by means of a `Solution` type.

Neighbourhoods

We assume that the exploration of the search space relies on the concept of *neighbourhood structure*, which has a central role in ROAs. A **neighbourhood structure** is given by the definition of a neighbouring relation between solutions. That is, $N(s, s')$ is a Boolean, whose value depends on whether s' can be obtained from s by means of the application of a small change operator, called **move**. The set of solutions s' that can be reached in this way from s forms the **neighbourhood** set of s, $N(s) \subseteq S$. **Neighbourhood structures can be defined for both complete and partial solutions**.

In this specification, the neighbourhood structure is specified by means of a `Move` type and a `Neighbourhood` type, this latter implementing the generation of moves.

Solution evaluation

The specification provides operations to query the feasibility and the objective value of candidate solutions. We do not assume to have available a mathematical description of F nor f, that is, we assume a **black box** scenario. Even if a mathematical description of the decision space is known, we assume that the feasibility and the objective value of **candidate solutions are assessed in an oracle manner**, that is, without exposing how these assessments have been computed; e.g., the feasibility and the objective value of candidate solutions could be assessed by executing a computationally costly simulation, but this information is not shared. However, when present, mathematical descriptions can be exploited to improve the implementation of the specification.

In ROA, the **objective function** is frequently replaced by an **evaluation function** g, which is also a mapping of the decision space S to a set of real numbers $\mathbb{R}$. The evaluation function can account for the objective function but also for penalties due to the violation of the problem constraints.

Types

The types to be defined by the user in this specification are:

- Problem
- Solution
- Move
- Neighbourhood

Problem

Signature

```
Problem
```

Description

The type `Problem` specifies the data structure to represent the particular *instance* of the problem to solve. An instance of a problem is the definition of the data related to the specific example of the abstract problem to solve. Data can be numerical or categorical values and specify, for example, the size of solutions, the presence or not of particular constraints, the values of coefficients in mathematical constraints or objectives.

Use cases

In the travelling salesman problem, `Problem` is a data structure containing at least the number of nodes to visit and the matrix of distances between the nodes. If the instance is geographical then the distance matrix can be replaced by the GPS coordinates of the nodes and by the formula to calculate the distance (Euclidean, haversine).

If the problem is finding the inputs to a simulator that yield the desired results then the instance may be the number of parameters input to the simulator and their domain. If these parameters are conditioned over other parameters then the values of the conditioning parameters must be defined in `Problem`.

See also

Solution, Neighbourhood, empty_solution, random_solution, construction_neighbourhood, destruction_neighbourhood, local_neighbourhood.

Candidate Solution

Signature

```
Solution
```

Description

The type `Solution` is a data structure defining a particular candidate solution. Therefore, `Solution` is an expression of the decision space. In case of combinatorial optimisation, `Solution` can be a particular combinatorial structure.

`Solution` instances are always associated to a problem instance, hence they have a reference to one such an instance. Additionally, associated to a `Solution` instance there is its evaluation value, that we assume to be of `float` type.

Candidate solutions define the elementary decisions (or *components*) from the decision space. For example, in a problem that asks to determine the value of a set of decision variables taken from their domains an elementary decision is the value assigned to a single variable. Candidate solutions can be restricted to be *complete*, in that all its elementary decisions are defined. Alternatively, they can be allowed to include *partial* solutions where only a subset of components is defined. This distinction is not always relevant. For some problems where it makes sense for candidate solutions to be defined as sets the number of components is not known a priori. Sets can also be represented by indicator vectors in which case the distinction is again relevant but the vector representation might not be convenient for the specific problem.

Instances of `Solution` must be candidate solutions but do not need to be *feasible*. For example, for a local search that works on complete solutions, these solutions do not need to satisfy all constraints of the problem but they do satisfy the requirement that all decisions are defined, even if they may break some other problem constraints. On the other hand, partial solutions would not be *valid* instances of `Solution` in that they do not satisfy the internal requirement of candidate solutions.

Candidate solutions can be direct or indirect representations of solutions. Indirect solution representations can be used when we can identify a smaller decision space, such that for a given member of this space a best corresponding solution for the original space can be derived in polynomial time. A somehow related distinction is done in evolutionary algorithms between genotype and phenotype. In this case, candidate solutions represent commonly genotypes.

Note, particular information used by the optimisation algorithm - tabu lists, pheromone matrix, etc - must be stored on the algorithm's side.

However, `Solution` may contain auxiliary data structures to facilitate calculations, for example, of increments.

Use cases

All algorithms need to be able to define, take as input and return as output instances of type `Solution`. Moreover, they need to be able to modify them via neighbourhoods and their moves.

See also

Problem, Neighbourhood, Move, empty_solution, random_solution, copy_solution, lower_bound, objective_value, moves, apply_move.

Neighbourhood

Signature

`Neighbourhood`

Description

The `Neighbourhood` type represents a particular neighbourhood structure defined over the decision space of a given problem instance. Neighbourhood structures are constructive or destructive if they work with partial candidate solutions and local if they work with complete solutions. Constructive neighbourhood structures consist of changes that add components to partial candidate solutions yielding partial or complete solutions. Destructive neighbourhoods structures consist of changes that remove components from complete or partial solutions yielding partial solutions. Local neighbourhood structures consist of changes that leave solutions complete.

Formally, a neighbourhood structure $\mathcal{N}$ can be defined as:

- a function $N : S_\pi \rightarrow 2^{S_\pi}$
- a function $N : S_\pi \times S_\pi \rightarrow T, F$
- a subset of pairs of candidate solutions $N \subseteq S_\pi \times S_\pi$
- a subset of candidate solutions for every solution s: $N(s) := s' \in S | N(s, s')$

Neighbourhoods are also characterised by:

- their size defined as $|N(s)|$
- symmetricity if $s' \in N(s) \implies s \in N(s')$
- neighbourhood graph of (S, N, π) a directed graph: $G_N := (V, A)$ with $V = S$ and $(uv) \in A \iff v \in N(u)$ (if symmetric neighbourhood then undirected graph)

Use cases

The algorithms will need to access the `Neighbourhood` instances associated with the problem. The related operations `construction_neighbourhood`, `destruction_neighbourhood` and `local(-search)-neighbourhoods` are used by algorithms to determine the instances of the `Neighbourhood` types implemented.

See also

Problem, Solution, Move, construction_neighbourhood, destruction_neighbourhood, local_neighbourhood, moves.

Move

`Move`

Description

The `Move` type identifies the changes between two neighbouring solutions. Formally, for a problem instance π, candidate solution space S_π, and a neighbourhood structure $\mathcal{N}(\pi) \subseteq (S \times S)$, there is an instance of `Move` for every pair of candidate solutions, $s, s' \in S$, s, s' in $\mathcal{N}(\pi)$ containing the information used by an operator function δ that applied to s yields s', that is, $s' = \delta(s)$. The operation `apply` implements the operator function.

It follows that a particular neighbourhood structure $\mathcal{N}(\pi) \subseteq (S \times S)$ can be fully represented by a collection of operator functions Δ and that for each candidate solution $s \in S$ the neighbourhood set $N(s)$ associated to $\mathcal{N}$ is generated by a subset of operator functions $\Delta(s) \subseteq \Delta$. Hence, $s' \in N(s) \iff s' = \delta(s), \delta \in \Delta(s)$. For each solution s the operator functions $\delta \in \Delta(s)$ are described by a set of instantiations of `Moves` that is generated by move generators, like the operations `moves`.

As an example, for candidate solutions that are linear permutations, a possible neighbourhood structure defines as neighbouring two solutions if they differ only in the position of two adjacent elements in the permutation. This neighbourhood structure can be represented by a `Move` with information about the position of the first element to swap. The operator function that uses this information will change the element indexed by `Move` with the following element in the permutation. The generator `moves` will instantiate all moves that are needed to reach all neighbouring solutions. In this case, there are as many moves as are the elements in the permutation each specifying a different index.

Use cases

Each neighbourhood structure defines its own moves that are generated by move generators such as `moves` and are used by the operator `apply` to implement the changes to the solution. Hence, we need a different definition for `Move`, `moves` and `apply` for each neighbourhood structure.

See also

Neighbourhood, Solution, moves, apply_move, invert_move, lower_bound_increment, objective_value_increment.

Operations

Empty solution

Signature

```
empty_solution(Problem) : Solution
```

Description

This function produces an empty solution for the given problem instance.

Use cases

Empty solutions are typically used as initial solutions in constructive search.

See also

Problem, Solution.

Heuristic solution

Signature

```
heuristic_solution(Problem) : Solution[0..1]
```

Description

This function produces a feasible solution for the given problem instance or none if the underlying heuristic fails to generate such a solution.

Use cases

Heuristic solutions are often used as initial solutions in local search.

See also

Problem, Solution.

Random solution

Signature

```
random_solution(Problem) : Solution
```

Description

This function samples the feasible decision space of the given problem instance uniformly at random (with replacement) and produces a feasible solution.

Use cases

Random solutions are often used as initial solutions in local search. The initial population of evolutionary algorithms typically consists of solutions generated at random.

See also

Problem, Solution.

Copy solution

Signature

```
copy_solution(Solution) : Solution
```

Description

This function produces a copy of the given solution.

Use cases

Making copies of solutions may be required when keeping track of the best solution found so far. It is also required by optimisation algorithms such as beam search and evolutionary algorithms.

See also

Solution.

Lower bound

Signature

```
lower_bound(Solution) : double[0..1]
```

Description

This function produces a lower bound on the value taken by the objective function at any feasible solution that can be obtained by applying construction moves to the given, presumably partial, solution. If it is known that no feasible solution can be obtained by further construction, this function should produce no value.

Evaluation of the lower bound must occur before this function returns, but the time at which the evaluation actually occurs is otherwise unspecified.

It is assumed that the objective function is to be minimised.

Use cases

Lower bounds are typically used in constructive search to guide solution construction or to stop it early (also known as *pruning*) by signalling that a better feasible solution than the best one known so far can no longer be constructed from a given partial solution.

Lower-bound functions are strongly related to the notion of *admissible heuristics* in computer science.

See also

Solution, Neighbourhood, Move, objective_value, empty_solution, apply_move, lower_bound_increment.

Objective value

Signature

```
objective_value(Solution) : double[0..1]
```

Description

This function produces the value of the objective function at the given solution if the solution is feasible, and no value otherwise. A solution is feasible if it satisfies all constraints of the problem. The objective value of an infeasible solution is undefined.

In general, both complete and partial solutions may be feasible or infeasible. Feasible partial solutions are common, for example, in models of the knapsack problem, as even an empty knapsack is a feasible solution. Infeasible complete solutions may arise, for example, when constructing solutions for the travelling salesman problem on incomplete graphs.

Evaluation of the objective function must occur before this function returns, but the time at which the evaluation actually occurs is otherwise unspecified.

It is assumed that the objective function is to be minimised.

Use cases

Objective-value evaluation is required by all optimisation algorithms.

See also

Solution, lower_bound, objective_value_increment.

Construction neighbourhood

Signature

```
construction_neighbourhood(Problem) : Neighbourhood
```

Description

This function returns the construction neighbourhood structure of the given problem instance. In a constructive-search model of a combinatorial optimisation problem, the construction neighbourhood structure, or construction rule, specifies how partial solutions, including empty solutions, may be made progressively more complete until a complete solution is reached.

Use cases

A construction neighbourhood structure is required by all constructive search approaches, including but not limited to greedy construction and ruin and recreate.

See also

Problem, Neighbourhood, destruction_neighbourhood, empty_solution.

Destruction neighbourhood

Signature

```
destruction_neighbourhood(Problem) : Neighbourhood
```

Description

This function returns the destruction neighbourhood structure of the given problem instance. In a constructive-search model of a combinatorial optimisation problem, the destruction neighbourhood structure, or destruction rule, dictates how complete or partial solutions may be made progressively less complete until an empty solution is reached.

Use cases

A destruction neighbourhood structure is required by some constructive search approaches, such as ruin and recreate.

See also

Problem, Neighbourhood, construction_neighbourhood, empty_solution.

Local neighbourhood

Signature

```
local_neighbourhood(Problem) : Neighbourhood
```

Description

This function returns the local neighbourhood structure of the given problem instance. In a local-search model of a combinatorial optimisation problem, the local neighbourhood structure specifies which feasible solutions, or neighbours, can be obtained from each feasible solution by means of a "small" modification. The notion of local optimum is intrinsically tied to the definition of such a neighbourhood.

Use cases

A local neighbourhood structure is required by all local search approaches, including but not limited to best-improvement, first-improvement and iterated local search, tabu search and evolutionary algorithms.

See also

Problem, Neighbourhood.

Moves

Signature

```
moves(Neighbourhood, Solution) : Move[0..*]
```

Description

This function provides for the complete enumeration of the neighbours of a given solution by producing a sequence of moves in unspecified order.

Pre-requisites

Both the given neighbourhood structure and the given solution must pertain to the same problem instance.

Use cases

Move enumeration is appropriate when the optimisation algorithm performs a full exploration of the set of neighbours of a solution before deciding how to proceed. In this case, enumeration order is irrelevant to the algorithm.

Greedy construction with random tie breaking and best-improvement local search are examples of algorithms where the whole set of neighbours is explored before the next move is accepted or the algorithm stops.

See also

Neighbourhood, Solution, Move, random_moves_without_replacement, apply_move.

Random move

Signature

```
random_move(Neighbourhood, Solution) : Move[0..1]
```

Description

This function provides for the random sampling of the neighbours of a given solution by producing a move drawn uniformly at random from the set of possible moves for that solution under the given neighbourhood structure or none if no such moves exist.

Pre-requisites

Both the given neighbourhood structure and the given solution must pertain to the same problem instance.

Use cases

Sampling moves uniformly at random is appropriate when only a partial exploration of the set of neighbours of a solution is performed by the optimisation algorithm. In this case, complete exploration of that set is neither expected nor detected.

Randomised local search (RLS) and most evolutionary algorithms are examples of algorithms where a single move for each solution is typically generated at random and immediately applied.

See also

Neighbourhood, Solution, Move, moves, apply_move.

Random moves without replacement

Signature

```
random_moves_without_replacement(Neighbourhood, Solution) : Move[0..*]
```

Description

This function provides for the complete enumeration of the neighbours of a given solution by producing a sequence of moves in random order.

Pre-requisites

Both the given neighbourhood structure and the given solution must pertain to the same problem instance.

Use cases

Sampling at random without replacement is appropriate when the optimisation algorithm explores the set of neighbours of a solution sequentially and may decide to stop the exploration and proceed some other way after seeing each move. In this case, the order in which moves are presented may influence how long such a partial exploration takes. Sampling at random without replacement avoids the bias inherent to deterministic enumeration, while still allowing completion of the exploration to be detected.

First-improvement local search is an example of an algorithm where moves are typically accepted before the whole set of neighbours is explored, but complete exploration is still performed in case no improving move is found.

See also

Neighbourhood, Solution, Move, moves, apply_move.

Apply move

Signature

```
apply_move(Move, Solution) : Solution
```

Description

This function applies a move to a solution in order to produce the corresponding neighbour.

Pre-requisites

The given move must have been generated under some neighbourhood for the given solution or a pristine copy of it, or be the inverse of the move that produced the given solution.

Use cases

Applying moves to solutions is required by all optimisation algorithms in order to visit new solutions.

See also

Move, Solution, invert_move, moves.

Invert move

Signature

```
invert_move(Move) : Move
```

Description

This function produces the inverse of a given move. If a given move can be applied to a solution A to obtain solution B, then applying its inverse to solution B must produce solution A. As a consequence, the inverse of a construction move must be a valid destruction move and vice-versa.

Use cases

Obtaining the inverse of moves allows backtracking to be performed from a given solution by applying the inverse of each move previously applied to the solution in reverse order. This assumes that storing, inverting and applying moves is generally more efficient than copying and storing solutions.

See also

Move, apply_move.

Lower-bound increment

Signature

```
lower_bound_increment(Move, Solution) : double[0..1]
```

Description

This function produces the difference between the lower bound of the solution that would be obtained by applying the given move to the given solution and the lower bound of the given solution. If lower bound of either solution is undefined, the lower-bound increment is also undefined, and this function produces no value.

Since it is assumed that the objective function is to be minimised, the lower-bound increment resulting from a construction move cannot be negative, and that resulting from a destruction move cannot be positive.

Pre-requisites

The given move must have been generated under some neighbourhood structure for the given solution or a pristine copy of it, or be the inverse of the move that produced the given solution.

Use cases

The lower-bound increment provides a heuristic measure of move quality in constructive search. In algorithms such as GRASP, construction moves typically consist of adding components to the current solution, and move quality is often seen as an attribute of the components themselves. The lower-bound increment provides a more general, drop-in replacement for such move-quality heuristics in constructive search.

Determining the lower-bound increment can often be performed faster than applying a move to a solution and computing the difference between the two lower-bound values. This avoids spending time applying moves to solutions that will be discarded immediately, and motivates further investment in efficient lower-bound increment evaluation, even if at the expense of some additional processing when moves are applied to solutions.

See also

Solution, Neighbourhood, Move, lower_bound, objective_value, objective_value_increment.

Objective-value increment

Signature

```
objective_value_increment(Move, Solution) : double[0..1]
```

Description

This function produces the difference between the value of the objective function at the solution that would be obtained by applying the given move to the given solution and the corresponding value at the given solution. If either solution is infeasible, the objective-value increment is undefined, and this function produces no value.

It is assumed that the objective function is to be minimised.

Pre-requisites

The given move must have been generated under some neighbourhood structure for the given solution or a pristine copy of it, or be the inverse of the move that produced the given solution.

Use cases

The objective-value increment provides a measure of move quality, especially in local-search algorithms, but also in constructive search algorithms.

Determining the objective-value increment can often be performed faster than applying a move to a solution and computing the difference between the two objective values. This avoids spending time applying moves to solutions that will be discarded immediately, and motivates further investment in efficient objective-value increment evaluation, even if at the expense of some additional processing when moves are applied to solutions.

See also

Solution, Neighbourhood, Move, objective_value, apply_move.

Glossary

complete solution

A solution for which all (decision) components are decided on.

construction neighbourhood

A neighbourhood structure for partial solutions where every neighbour solution is more complete, in the sense that more components are decided on, and may either be a partial or a complete solution. This means that, the consecutive application of (constructive) moves to a given partial solution will eventually lead to a complete solution.

decision space

The domain of the optimisation problem, which contains the set of all partial and all complete (candidate) solutions for the given problem definition.

destruction neighbourhood

A neighbourhood structure for partial and for complete solutions where every neighbour solution is less complete, in the sense that less components are decided on, and is a partial solution. This means that, the consecutive application of (destructive) moves to a given solution will eventually lead to an empty solution.

empty solution

A solution for which no components are decided on.

feasible set

The subset of the decision space consisting of all feasible solutions.

feasible solution

A solution for which the objective function is defined.

infeasible solution

A solution for which the objective function is undefined.

local neighbourhood

A neighbourhood structure for solutions that are complete and feasible, and where each neighbour solution is also complete and feasible.

move

A description of, or a data structure encoding, a set of changes to be applied to a solution to obtain a neighbour solution. A move is assumed to always be associated to some neighbourhood structure.

neighbour

A solution that, under a given neighbourhood structure, can be obtained by applying a set of changes (a *move*) to a given solution, in which case the former solution is called a neighbour of the latter.

neighbourhood size

The total number of neighbours of a given solution under a given neighbourhood structure.

neighbourhood structure

A description of the neighbourhood of any given solution, which relies on a set of rules that define which solutions are neighbours of the given solution, and the *moves* that lead to them.

objective function

A function mapping a solution in the decision space to an element of the objective space (in this case, a real value). Minimisation is assumed, that is, a solution is considered better than any other solution with a greater objective value.

objective space

The codomain of the objective function. In this specification, the objective space is the set of real

values, $\mathbb{R}$.

partial solution

A solution for which some of the components are not decided on.

solution

An element of the decision space. A solution is described by a set of (decision) components, which may or may not be decided on.