SIGEVO Summer School (S3)

Modelling Optimization Problems for Evolutionary Algorithms and Other Metaheuristics

Project statement

Carlos M. Fonseca
Department of Informatics Engineering
University of Coimbra
Portugal

21 June 2022

1 Introduction

This project consists in modelling given combinatorial optimization problems either as local-search or constructive-search problems and implementing the corresponding models while bearing in mind the principles and ideas presented in the lecture. Problem implementations shall adhere to suitable Application Programming Interfaces (APIs), so that a number of simple algorithms, or *solvers* (provided together with the API specifications), can run directly on them in a problem-independent way, and experimental results can be collected.

Three sub-projects are proposed based on two different problems. Each sub-project consists in modelling and implementing one of the problems in either C (the preferred language) or Python3 (for one of the two problems only) by a group of 3 to 5 students (typically 4). It is essential that most, but not necessarily all, members of each group are comfortable programming (all in the same language), 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 on the last day of the School. Presentations should cover a description of the problem considered, modelling decisions, such as solution representation, neighbourhood structure and/or construction rules, implementation decisions, including search space sampling, search operators, and solution evaluation strategies, as well as (preliminary) experimental results.

1.1 Software requirements

The required development environment depends on the language of choice:

C A modern C compiler such as a recent version of `gcc` or `clang` is required. The ability to compile against the GNU Scientific Library (v2.5 or later) is also required, although this requirement can be worked around in case it proves difficult to meet.

In addition, development tools such as `make` (build automation), a debugger, and `valgrind` (dynamic analysis) will be very useful, but are not strictly required.

Python3 A Python3 installation containing the numerical package `Numpy` is required.

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

1.2 Programming skills

The C APIs use Abstract Data Types (ADTs) implemented as pointers to structures. Students should be familiar with structures, arrays, and pointers in C.

Python3 implementations will consist of concrete classes that derive from a provided set of abstract classes, and Numpy arrays are used throughout. Students should be familiar with object-oriented programming in Python and with Numpy. Numpy array arithmetic, indexing and universal functions (“ufuncs”) are essential for the problems modelled in Python to achieve reasonable performance.

2 Problem descriptions

2.1 3D printing service

Programming language: Python or C

An additive manufacturing startup provides 3D printing services both to 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, the machine becomes available after a fixed change-over time. Each part must be ready for delivery 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 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.

2.2 Community detection

Programming language: C

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.

3 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?

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.

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).

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.

3.1 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. **[Local search]** Solutions can be easily modified to obtain neighbouring solutions?
3. **[Constructive search]** Feasible solutions can be easily constructed by successively adding components?

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.,

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

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

4 Application Programming Interfaces

4.1 C local-search API

This is (a subset of) the nasf4nio API with a single modification related to problem instantiation. In your project, you should use the provided `s3problem.h` header file instead of `problem.h`.

The symbols marking the items of these interfaces have the following meaning:

- Basic elements required to support simple random search.

▷ Additional elements required to support a simple mutation-only evolutionary algorithm.

◇ Further elements required to support an iterated local-search algorithm.

4.1.1 Data structures

- `struct problem {...}`
- `struct solution {struct problem *p;...}`

▷ `struct move {struct problem *p;...}`

4.1.2 Problem instantiation and inspection

- `struct problem *newProblem(char *filename) {...}`
  Allocate a problem structure and initialize it with the problem-specific data contained in the specified input file. The function returns a pointer to the allocated problem structure on success and NULL on error.

- `int getNumObjectives(const struct problem *p) {return 1;}`

4.1.3 Memory management

- `void freeProblem(struct problem *p) {...}`
  Free all memory used by a problem structure.

- `struct solution *allocSolution(struct problem *p) {...}`
  Allocate memory for a solution and return a pointer to it. Return NULL if allocation fails.

- `void freeSolution(struct solution *s) {...}`
  Free all memory used by a solution structure.

▷ `struct move *allocMove(struct problem *p) {...}`
Allocate memory for a move and return a pointer to it. Return NULL if allocation fails.

▷ `void freeMove(struct move *v) {...}`
Free all memory used by a move structure.

4.1.4 Reporting

- `void printProblem(struct problem *p) {...}`
  Print a user-formatted representation of a problem instance.

- `void printSolution(struct solution *s) {...}`
  Print a user-formatted representation of a solution.

▷ `void printMove(struct move *v) {...}`
  Print a user-formatted representation of a move.

4.1.5 Operations on solutions

- `struct solution *randomSolution(struct solution *s) {...}`
  Uniform random sampling of the solution space.
  The input argument must be a pointer to a solution previously allocated with `allocSolution()`, which is modified in place.
  If `randomMoveWOR()` is implemented, this function must also initialize the corresponding state by performing the equivalent to `resetRandomMoveWOR()`.
  The function returns its first input argument.

- `struct solution *copySolution(struct solution *dest, const struct solution *src) {...}`
  Copy the contents of the second argument to the first argument, which must have been previously allocated with `allocSolution()`. The function returns its first input argument.

- `double *getObjectiveVector(double *objv, struct solution *s) {...}`
  Single or multiple objective full and/or incremental solution evaluation. Once a solution is evaluated, results may be cached in the solution itself so that a subsequent call to this function simply returns the precomputed value and/or the solution can be re-evaluated more efficiently after it is modified by one or more calls to `applyMove()`. Therefore, the formal argument is not `const`. Solution (re-)evaluation must occur before this function returns, but the time at which actual evaluation occurs is otherwise left unspecified. The function returns its first input argument.

▷ `struct solution *applyMove(struct solution *s, const struct move *v) {...}`
  Modify a solution by applying a move to it. It is assumed that the move was generated for, and possibly evaluated with respect to, that particular solution. The result of applying a move to a solution other than that for which

it was generated/evaluated (or a pristine copy of it), including applying the same move to a solution more than once, is undefined.
If `randomMoveWOR()` is implemented, this function must also initialize the corresponding state by performing the equivalent to `resetRandomMoveWOR()`. The function returns its first input argument.

◇ `int getNeighbourhoodSize(struct solution *s) {...}`
Return the number of neighbours of a given solution.

◇ `struct solution *resetRandomMoveWOR(struct solution *s) {...}`
Reset the uniform random sampling without replacement of the neighbourhood of a given solution, so that any move can be generated by the next call to `randomMoveWOR()`. The function returns its input argument.

4.1.6 Operations on moves

▷ `struct move *randomMove(struct move *v, const struct solution *s) {...}`
Uniform random sampling of the neighbourhood of a given solution, with replacement. The first input argument must be a pointer to a move previously allocated with `allocMove()`, which is modified in place.

▷ `struct move *copyMove(struct move *dest, const struct move *src) {...}`
Copy the contents of the second argument to the first argument, which must have been previously allocated with `allocMove()`. The function returns its first input argument.

◇ `double *getObjectiveIncrement(double *obji, struct move *v, struct solution *s) {...}`
Single or multiple objective move evaluation with respect to the solution for which it was generated, before it is actually applied to that solution (if it ever is). The result of evaluating a move with respect to a solution other than that for which it was generated (or to a pristine copy of it) is undefined. Once a move is evaluated, results may be cached in the move itself, so that they can be used by applyMove() to update the evaluation state of the solution more efficiently. In addition, results may also be cached in the solution in order to speed up evaluation of future moves. Consequently, neither formal argument is const. The function returns its first input argument.

◇ `struct move *randomMoveWOR(struct move *v, struct solution *s) {...}`

Uniform random sampling of the neighbourhood of a given solution, without replacement. The first input argument must be a pointer to a move previously allocated with `allocMove()`, which is modified in place. The function returns this pointer if a new move is generated or NULL if there are no moves left.

4.2 C constructive-search API

This is (a subset of) the constructive-search API proposed by Outeiro (2021). It is similar to the nasf4nio API in many aspects, but it it *not* compatible with it. The development of an API supporting both local search and constructive search in an integrated manner is ongoing work.

Only the basic elements required to support a simple version of greedy randomized constructive search are described here. In particular, only the ADD subneighbourhood will be used.

4.2.1 Data structures

- `struct problem {...}`
- `struct solution {struct problem *p;...}`
- `struct component {struct problem *p;...}`

4.2.2 Problem instantiation and inspection

- `struct problem *newProblem(char *filename) {...}`
  Allocate a problem structure and initialize it with the problem-specific data contained in the specified input file. The function returns a pointer to the allocated problem structure on success and NULL on error.

- `long getMaxNeighbourhoodSize(struct problem *p, const enum SubNeighbourhood nh) {...}`
  Return the largest possible number of direct neighbours of any solution in a given subneighbourhood (or a larger number). The result may be cached in the problem in order to speed up future operations. Therefore, the first input argument is not `const`.

- `int getNumObjectives(const struct problem *p) {return 1;}`

4.2.3 Memory management

- `void freeProblem(struct problem *p) {...}`
  Free all memory used by a problem structure.

- `struct solution *allocSolution(struct problem *p) {...}`
  Allocate memory for a solution and return a pointer to it. Return NULL if allocation fails.

- `void freeSolution(struct solution *s) {...}`
  Free all memory used by a solution structure.

- `struct move *allocComponent(struct problem *p) {...}`
  Allocate memory for a solution component and return a pointer to it. Return NULL if allocation fails.

- `void freeComponent(struct move *v) {...}`
  Free all memory used by a component structure.

4.2.4 Reporting

- `void printProblem(struct problem *p) {...}`
  Print a user-formatted representation of a problem instance.

- `void printSolution(struct solution *s) {...}`
  Print a user-formatted representation of a solution.

- `void printComponent(struct component *c) {...}`
  Print a user-formatted representation of a solution component.

4.2.5 Operations on solutions

- `struct solution *emptySolution(struct solution *s) {...}`
  Initialize a solution structure as an empty solution. The input argument must be a pointer to a solution previously allocated with `allocSolution()`, which is modified in place. If `enumMove()` is implemented, `emptySolution()` must also reset the corresponding state by performing the equivalent to `resetEnumMove()`. The function returns its input argument.

- `struct solution *resetEnumMove(struct solution *s, const enum SubNeighbourhood nh) {...}`
  Reset the enumeration of a given subneighbourhood of a solution, so that the next call to `enumMove()` will start the enumeration of that subneighbourhood from the beginning. The function returns its input argument.

- `struct solution *copySolution(struct solution *dest, const struct solution *src) {...}`
  Copy the contents of the second argument to the first argument, which must have been previously allocated with `allocSolution()`. The function returns its first input argument.

- `double *getObjectiveVector(double *objv, struct solution *s) {...}`
  Single or multiple objective full and/or incremental solution evaluation. This function modifies the array passed as first input argument to contain the objective values. Once a solution is evaluated, results may be cached in the solution itself so that a subsequent call to this function simply returns the precomputed value and/or the solution can be re-evaluated more efficiently after it is modified by one or more calls to `applyMove()`. Therefore, the formal argument is not `const`. Solution (re-)evaluation must occur before this function returns, but the time at which actual evaluation occurs is otherwise left unspecified. The function returns its first input argument if the given solution is feasible or NULL if it is infeasible.

- `double *getObjectiveLB(double *objLB, struct solution *s) {...}`
  Single or multiple objective full and/or incremental lower bound evaluation. This function modifies the array passed as first input argument to contain the lower bound values. Once the lower bound of a solution is evaluated, results may be cached in the solution itself so that a subsequent call to this function simply returns the precomputed value(s) and/or the solution can be re-evaluated more efficiently after it is modified by one or more calls to `applyMove()`. Therefore, the formal argument is not `const`. The function returns its first input argument.

▷ `struct solution *applyMove(struct solution *s, const struct component *c, const enum SubNeighbourhood nh) {...}`
  Modify a solution by applying a move to it. It is assumed that the move was generated for, and possibly evaluated with respect to, that particular solution and the given subneighbourhood. In addition, once a move is applied to a solution, it can be reverted by applying it again with the opposite subneighbourhood. For example, after an ADD move generated for a given solution is applied to that solution, it may be applied again as a REMOVE move to the resulting solution in order to recover the original solution. The result of applying a move to a solution in any other circumstances is undefined. If `enumMove()` is implemented, `applyMove()` must also reset the corresponding state by performing the equivalent to `resetEnumMove()`. The function returns its first input argument.

4.2.6 Operations on moves

- `double *getObjectiveLBIncrement(double *obji, struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Single or multiple objective move evaluation with respect to the solution and subneighbourhood for which the move was generated, before it is actually applied to that solution (if it ever is). The result of evaluating a move with respect to a solution and subneighbourhood other than those for which it was generated (or to a pristine copy of that solution and the same neighbourhood) is undefined. Once a move is evaluated, results may be cached in the component so that they can be used by `applyMove()` to update the evaluation state of the solution more efficiently. In addition, results may also be cached in the solution in order to speed up future operations. Consequently, neither formal argument is `const`. The function updates and returns its first input argument, which may have been modified (in particular, to store the computed lower bound increments).

- `struct component *enumMove(struct component *c, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Enumeration of a given subneighbourhood of a solution, in an unspecified order, that allows for fast neighbourhood exploration and evaluation with `getObjectiveLBIncrement()`, particularly when a large part of the neighbourhood is to be visited. The first input argument must be a pointer to a component previously allocated with `allocComponent()`, which is modified in place. The function returns this pointer if a new move is generated or NULL if there are no moves left.

4.3 Python3 API

4.3.1 Problem class

A problem is implemented as a single class that defines two inner classes and a number of "standard" methods. The high level structure of a problem class definition is as follows:

```
import numpy as np
import sample

class ProblemX:
    class Solution(sample.Element):
        ...
    class SolutionSample(sample.Sample):
```

```
        ...
    class Move(sample.Element):
        ...
    class MoveSample(sample.Sample):
        ...
    def __init__(self, ...):
        ...
    def randomSolution(self, n=None):
        if n is None:
            # return a single solution
            return self.Solution(self, ...)
        else:
            ...
            # return a sample of n solutions
            return self.SolutionSample(self, ...)
```

Classes `Solution` and `Move` are used to store single solutions and single moves, respectively. Classes `SolutionSample` and `MoveSample` are containers for sequences of solutions and moves, respectively, and should emulate part of the behaviour of 1-dimensional Numpy arrays. In particular, indexing a `SolutionSample` object with an integer should return a `Solution` object. `Solution` objects can be transformed into `SolutionSample` objects via a `.toSample()` method, and analogously for `Move` and `MoveSample` objects.

The `__init__` method implements problem-specific instantiation from user-supplied instance data, which is typically stored as attributes of the newly instantiated problem instance.

The `randomSolution` method implements random solution generation. Depending on the value of `n`, the generated solutions are returned as a newly instantiated `Solution` or `SolutionSample` object, as appropriate.

4.3.2 Solution classes

The `Solution` class should have the following structure:

```
class Solution(sample.Element):
    # argument 'of' is the outer problem instance object
    def __init__(self, of, data, ...):
        self.of = of
        self.sample = of.SolutionSample
        ...
    # make a copy of the current solution
    def copy(self):
```

```
        return self.__class__(self.of, ...)
    # toSample() should return a "view" of the current
    # solution data, not a copy
    def toSample(self):
        return self.sample(self.of, ...)
    # apply single move 'other' to current solution, in place
    def __iadd__(self, other):
        ...
    # apply move(s) to current solution
    def __add__(self, other):
        ... # return a Solution if 'other' is a Move object, or a
            # SolutionSample object if it is a MoveSample object
    # random move generation
    def randomMove(self, n=None):
        if n is None:
            ... # return a Move object
        else:
            ... # return a MoveSample object
    # objective function evaluation
    def objvalue(self):
        ... # return a scalar
```

The `SolutionSample` class should implement essentially the same methods as the `Solution` class, except `toSample`. Additional methods and any differences with respect to the corresponding `Solution` methods are detailed next:

```
class SolutionSample(sample.Sample):
    ...
    # apply moves to current solutions
    def __add__(self, other):
        ... # return a SolutionSample object
    # random move generation
    def randomMove(self, n=None):
        ... # return a MoveSample object
    # emulate 1-d array indexing/slicing
    def __getitem__(self, key):
        ...
    def __setitem__(self, key, value):
        ...
    # emulate the Numpy array repeat() method
    def repeat(self, n):
        return self.__class__(self.of, ...)
```

```
        # objective function evaluation
        def objvalue(self):
            ... # return a 1-dimensional Numpy array
```

4.3.3 Move classes

The `Move` and `MoveSample` classes can usually be implemented as follows, where `data` should contain the representation of the move(s).

```
    class Move(sample.Element):
        def __init__(self, of, data):
            self.of = of
            self.sample = of.MoveSample
            self.data = np.asarray(data)

    class MoveSample(sample.Sample):
        def __init__(self, of, data):
            self.of = of
            self.element = of.Move
            self.data = np.asarray(data)
```

In this case, the default methods inherited from the parent classes do not need to be redefined.

4.3.4 Additional remarks

In practice, there is usually no need to produce dedicated implementations for the the same methods of `Solution` and `SolutionSample`. In fact, provided that a given method is implemented for the latter, it can also be easily implemented for the former by promoting the solution to a sample using `.toSample()`, calling the desired method, and indexing the result to obtain a solution again. Alternatively, if the method is initially implemented for `Solution` only, a simple implementation for `SolutionSample` can be produced by iterating over `self` and successively calling the existing `Solution` method.

One last feature concerns operations involving solution samples and move samples of different sizes. Calling `randomMove` with a value of `n` greater than 1 will generate a move sample with n times the elements of the original solution sample. In this case, the first n moves correspond to the first solution, the next n moves correspond to the second solution, and so on. The `__add__` methods should operate in accordance to this.

The supplied implementation of the n-queens problem illustrates many of these aspects, and can be used as a reference.

References

Outeiro, S. B. (2021). An application programming interface for constructive search. Master's thesis, University of Coimbra, Portugal. https://hdl.handle.net/10316/98261