ALGO Code Fest #0

Modelling Optimisation Problems for Constructive Search Algorithms

Problem Statement

Samuel B. Outeiro and Carlos M. Fonseca
Department of Informatics Engineering
University of Coimbra
Portugal

July 28th, 2021

1 Introduction

In this Code Fest, the task assigned to participants consists in modelling given combinatorial optimisation problems as constructive search problems and implementing the corresponding models while bearing in mind the principles and ideas shown in the presentation. Problem implementations shall adhere to an Application Programming Interface (API) so that a number of constructive search algorithms, or *solvers* (provided together with the API specification), can run directly on them in a problem-independent way, and experimental results can be collected.

Five different problems are proposed. Each problem should be modelled and implemented by a group of 2 to 5 participants (preferably 3). It is essential that most, but not necessarily all, members of each group are comfortable programming in C, 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 small presentation at the end of the event. Presentations should cover a description of the problem con-

sidered, modelling decisions, such as solution representation, construction rules, and lower/upper bound formulations, implementation decisions, including move enumeration and objective function and lower/upper bound evaluation strategies, and (preliminary) experimental results.

1.1 Software Requirements

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.

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

1.2 Programming Skills

The API uses Abstract Data Types (ADTs) implemented as pointers to structures. Participants should be familiar with structures, arrays, and pointers in C.

2 Problem Descriptions

2.1 3D Printing Service

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 minimise the total penalty incurred by the service provider.

2.2 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 minimise the total (trench and cable) set up cost of the new network.

2.3 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 maximise the total internal edge weight of all groups.

2.4 Hypervolume Subset Selection

The Hypervolume Indicator of a set of points $S \subset \mathbb{R}^d$, $d > 0$, is the size of the region dominated by those points and bounded above (assuming minimisation) by a given reference point, $r \in \mathbb{R}^d$. Formally,

$$H(S) = \Lambda(\{q \in \mathbb{R}^d \mid \exists p \in S : p \leq q \wedge q \leq r\})$$

where $\Lambda(\cdot)$ denotes the Lebesgue measure, which is simply the length, area, and volume for $d = 1, 2, 3$, respectively.

Given such a point set, S, the Hypervolume Subset Selection problem consists of finding a subset $A \subseteq S$ of size up to a given value $k \leq |S|$ such that $H(A)$ is maximum.

2.5 Self-Driving Rides

A fleet of self-driving vehicles is to be assigned to a list of pre-booked rides, so that riders may get to their destination on time. These vehicles navigate a city that is represented by a grid of streets, and they have limited time to make their assigned rides. If a ride is completed on time, a reward is given

to the corresponding vehicle. Furthermore, there is an incentive if a vehicle starts a ride at a specified time.

Once a ride begins, the vehicle has to drive a distance that is defined as the Manhattan distance between two points, where a point represents an intersection in the grid of streets. Each distance unit travelled by a vehicle corresponds to a time unit.

The problem consists in assigning rides to vehicles such that the reward given by completing rides on time or starting rides at a specified time is maximised.

For more information, one may check a more detailed description here. Moreover, input data for this problem can be obtained here.

3 Problem Modelling

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

Problem instance What (known) data is required to fully characterise 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 characterise 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 optimisation 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 minimised or maximised?

Construction rules What can be seen as an *partial solution*, and how do such partial solutions relate to one another and to *feasible* solutions? In combinatorial optimisation, solutions can usually be understood as subsets of a larger set of solution *components* known as the *ground set*. Feasible solutions can thus be constructed by starting with the empty set and adding one component at a time according to certain model-specific rules. Partial solutions are the intermediate subsets of components that are created in such a process, and can also be seen as a representation of all feasible solutions that contain them. This allows the performance of partial solutions to be inferred from the sets of solutions which they represent.

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 can 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 can (possibly partial) solutions be represented so that:

1. Objective function values (where applicable) and related bounds can be computed efficiently?
2. 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*, i.e., the *addition* or *removal* of components to/from a solution, be represented?

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

Incremental solution evaluation When an evaluated solution is modified by applying one or more moves to it, can the resulting solution be evaluated faster? How?

Move evaluation How much would applying a given move to a 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?

3.2 Search operators

Having established how solutions and moves are represented and evaluated, the problem model needs to provide the elementary problem-dependent *operations* used by constructive search solvers. The following few operations are commonly used:

Empty solution generation is used for search initialisation

Heuristic solution generation is used to obtain tentatively good, feasible solutions from given partial solutions, including the empty solution. It can also be used initialise the search (seeding).

Move enumeration is used whenever the *construction neighbourhood* is to be completely explored, e.g., in order to select the best move to apply. Since the order of enumeration is irrelevant in the light of this, the most favourable order from a *computational* perspective is usually implemented.

Random move generation (with replacement) arises whenever neighbourhood exploration is limited to a small number of random neighbours. It is usually simpler and more efficient to implement than sampling without replacement.

Heuristic move generation (with or without replacement) favours potentially good moves, but the effect of such moves on upper/lower bounds not guaranteed, which is possibly simpler and more efficient to implement than generating truly greedy moves based on true bounds.

4 Application Programming Interface

This is an extension of the nasf4nio API for constructive search. Each group should implement their problem in the provided `problemX.c` C source file, in the corresponding `problem/problemX` directory.

The symbols marking the items of this interface have the following meaning:

- • Basic elements required to support greedy randomized search.
- ▷ Additional elements required to support ant colony optimisation.
- ◇ Further elements required to support an iterated greedy algorithm.

4.1 Data Structures

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

4.1.1 Problem Instantiation and Inspection

- `struct problem *newProblem(char *filename) {...}`
  Allocate a problem structure and initialise it with problem-specific data contained in a specified input file. The function returns a pointer to the problem if instantiation succeeds or `NULL` if it fails.

- `long getMaxNeighbourhoodSize(const struct problem *p, const enum SubNeighbourhood nh) {...}`
  Return the largest possible number of direct neighbours in a given sub-neighbourhood (or a larger number).

- ▷ `long getNumComponents(const struct problem *p) {...}`
  Return the size of the ground set of a problem instance.

- ▷ `long getMaxSolutionSize(const struct problem *p) {...}`
  Return the largest number of components that a solution can potentially have.

4.1.2 Memory Management

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

- `struct solution *allocSolution(struct problem *p) {...}`
  Allocate memory for a solution structure. The function returns a pointer to the solution if allocation succeeds or `NULL` if it fails.

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

- `struct move *allocMove(struct problem *p) {...}`
  Allocate memory for a move structure. The function returns a pointer to the move if allocation succeeds or `NULL` if it fails.

- `void freeMove(struct move *v) {...}`
  Deallocate the memory used by a move structure.

4.1.3 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.

4.1.4 Operations on Solutions

- `struct solution *emptySolution(struct solution *s) {...}`
  Initialise 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 any of the following functions:
  - `enumSolutionComponents()`
  - `enumMove()`

  is implemented, `emptySolution()` must also reset the corresponding state by performing the equivalent to:
  - `resetEnumSolutionComponents()`
  - `resetEnumMove()`

  respectively.
  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 copied solution is functionally indistinguishable from the original solution. 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 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(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 if a given solution is feasible or `NULL` if it is unfeasible, in which case the first input argument is left unspecified (in particular, it may have been modified).

- `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 lower bound values. The lower bound of a solution must be less than or equal to the lower bound of another solution if the sets of present and forbidden components of the former are both subsets of the corresponding sets of components of the latter. 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 move *v, const enum SubNeighbourhood nh) {...}`
  Modify a solution in place 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 any of the following functions:
  - `enumSolutionComponents()`
  - `enumMove()`

  is implemented, `applyMove()` must also reset the corresponding state by performing the equivalent to:
  - `resetEnumSolutionComponents()`
  - `resetEnumMove()`

  respectively, for all subneighbourhoods.
  The function returns its first input argument.

- `int isFeasible(struct solution *s) {...}`
  Return true if a given solution is feasible or false if it is unfeasible. The result may be cached in the solution in order to speed up future operations. Therefore, the input argument is not `const`.

- `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.

▷ `long getNeighbourhoodSize(struct solution *s, const enum SubNeighbourhood nh) {...}`
Return the number of direct neighbours in a given subneighbourhood of a solution. The result may be cached in the solution in order to speed up future operations. Therefore, the first input argument is not `const`.

▷ `long enumSolutionComponents(struct solution *s, const enum ComponentState st) {...}`
Enumerate the components of a solution that are in a given state, in unspecified order. The function returns a unique component identifier in the range $0..|\mathcal{G}|-1$ if a new component is enumerated, where $|\mathcal{G}|$ denotes the size of the ground set of the problem instance, or a negative integer if there are no components left.

▷ `struct solution *resetEnumSolutionComponents(struct solution *s, const enum ComponentState st) {...}`
Reset the enumeration of the components of a solution that are in a given state, so that the next call to `enumSolutionComponents()` will start the enumeration of the components in that state from the beginning. The function returns its first input argument.

◇ `struct solution *heuristicSolution(struct solution *s) {...}`
Heuristically construct a feasible solution, preserving all present and forbidden components in a given solution, which is modified in place.
If any of the following functions:

- `enumSolutionComponents()`
- `enumMove()`

is implemented, `heuristicSolution()` must also reset the corresponding state by performing the equivalent to:

- `resetEnumSolutionComponents()`
- `resetEnumMove()`

respectively.
The function returns its input argument if a new feasible solution is generated or `NULL` if no feasible solution is found, in which case the original input argument is lost.

4.1.5 Operations on Moves

- `struct move *enumMove(struct move *v, struct solution *s, const enum SubNeighbourhood nh) {...}`
  Enumeration of a given subneighbourhood of a solution, in an unspecified order. This is intended to support 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 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.

- `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 copied move is functionally indistinguishable from the original move. The function returns its first input argument.

- `double *getObjectiveLBIncrement(double *obji, struct move *v, struct solution *s, const enum SubNeighbourhood nh) {...}`
  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). This function modifies the array passed as first input argument to contain lower bound increment values. 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.

▷ `long getComponentFromMove(const struct move *v) {...}`
  Return the unique component identifier in the range $0..|\mathcal{G}|-1$ with respect to a given move, where $|\mathcal{G}|$ denotes the size of the ground set of the problem instance.

◇ `struct move *randomMove(struct move *v, struct solution *s, const enum SubNeighbourhood nh) {...}`

Uniform random sampling with replacement of a given subneighbourhood of a solution. This may be an empty set for some subneighbourhoods, in which case the function returns `NULL`. The first input argument must be a pointer to a move previously allocated with `allocMove()`, which is modified in place.

5 Additional Remarks

5.1 Compiling a Problem for a Range of Solvers

1. Compile utilities by running the `Makefile` in the `util` directory.

```
[nasf4nio-cs]$ cd src/util/
[util]$ make
```

2. Compile solvers by running the `Makefile` in the `solver` directory.

```
[nasf4nio-cs]$ cd src/solver/
[solver]$ make
```

3. Edit the `Makefile` in the problem model directory to select (uncomment) or deselect (comment) the solvers to apply to the problem. In the following example, *GRASP* is selected and *Iterated Greedy with Biased Destruction* is not selected.

```
[nasf4nio-cs]$ cd problem/problemX/
[problemX]$ more Makefile
(...)
PROG = problemX-grasp1
(...)
problemX-grasp1:  problemX.o
$(CC) -o $@ $(CFLAGS) $(OPT) $^ $(SDIR)/grasp1.o $(LIBS)
# problemX-bd_ig: problemX.o
# $(CC) -o $@ $(CFLAGS) $(OPT) $^ $(SDIR)/bd_ig.o $(LIBS)
(...)
```

4. Compile the problem model for the selected solvers and run them. In the following example, *GRASP* is applied to the problem.

```
[problemX]$ make
[problemX]$ ./problemX-grasp1 <input data> 0.3 1000
```

5.2 Default Parameter Values for Solvers

Greedy Randomized Adaptive Search Procedure (GRASP): The default values for the proportion of components in the restricted candidate list is $\alpha = [0, 1]$.

Ant Colony Optimisation (ACO) - *Ant Colony System* (ACS): The default population size is $m = 10$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default value for the evaporation rate in the local pheromone trail update rule is $\xi = 0.1$. The default value for the evaporation rate in the global pheromone trail update rule is $\rho = 0.1$. The default value for the probability used in the pseudorandom proportional action choice rule is $q_0 = 0.9$. The default initial pheromone value is $\tau_0 = 1/nC^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Ant Colony Optimisation (ACO) - *Ant System* (AS): The default population size is $m = n$, where n denotes the size of the problem instance. The default value for the relative influence of the pheromone trail is $\alpha = 1$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default value for the evaporation rate is $\rho = 0.5$. The default value for parameter Q is $Q = 1$. The default initial pheromone value is $\tau_0 = m/C^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Ant Colony Optimisation (ACO) - *Elitist Ant System* (EAS): The default population size is $m = n$, where n denotes the size of the problem instance. The default value for the relative influence of the pheromone trail is $\alpha = 1$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default value for the evaporation rate is $\rho = 0.5$. The default value for parameter Q is $Q = 1$. The default value for the weight given to the best-so-far solution is $e = n$. The default initial pheromone value is $\tau_0 = (m + e)/\rho C^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Ant Colony Optimisation (ACO) - *Hyper-Cube Framework* (HC-ACO): The default population size is $m = n$, where n denotes the size of the problem instance. The default value for the relative influence of the pheromone trail is $\alpha = 1$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default value for the evaporation rate is $\rho = 0.5$. The default value for parameter Q is $Q = 1$. The default initial pheromone value is $\tau_0 = m/C^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Ant Colony Optimisation (ACO) - $\mathcal{MAX} - \mathcal{MIN}$ *Ant System* ($\mathcal{MM}$AS):

The default population size is $m = n$, where n denotes the size of the problem instance. The default value for the relative influence of the pheromone trail is $\alpha = 1$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default value for the evaporation rate is $\rho = 0.02$. The default value for the probability of constructing the best solution is $p_{best} = 0.05$. The default initial pheromone value is $\tau_0 = m/\rho C^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Ant Colony Optimisation (ACO) - *Rank-Based Ant System* (AS_{rank}): The default population size is $m = n$, where n denotes the size of the problem instance. The default value for the relative influence of the pheromone trail is $\alpha = 1$. The default values for the relative importance of the heuristic information is $\beta = [2, 5]$. The default number of ants that lay pheromones is $w = 6$. The default value for the evaporation rate is $\rho = 0.1$. The default initial pheromone value is $\tau_0 = 0.5r(r-1)/\rho C^{nn}$, where C^{nn} denotes the objective value of a solution generated by a heuristic. However, the default initial pheromone can also be $\tau_0 = 1$.

Iterated Greedy (with uniform random sampling): The default destruction size is $d = 4$. The default temperature value is $T = 100$.

Iterated Greedy (with biased destruction): The default destruction size is $d = 4$. The default temperature value is $T = 100$.