COIN-OR METSlib
a Metaheuristics Framework
in Modern C++.

Mirko Maischberger

Dipartimento di Sistemi e Informatica,
Università degli Studi di Firenze
Via di S. Marta, 3 - 50139 Firenze
email: <mirko.maischberger@gmail.com>

April 23, 2011

(this document refers to version 0.5.2 of the library)

Contents

Chapter 1

Introduction

1.1 Introduction

As a software framework METS*lib* provides a common abstraction of local search derived metaheuristics algorithms. Once the user has defined her own data structures and operators, the framework allows an easy personalisation of the algorithms and an easy comparison and combination of different local search strategies. The behaviour of single aspects of each algorithm can be controlled implementing specific specialisations.

METS*lib* emphasise code reuse: problem, operators and neighbourhoods can be solved with all the provided algorithms without requiring changes in the model code. The model and the solver can and should evolve independently. Moreover METS*lib* code is fully re-entrant and can be used to run more searches in parallel without problems.

The library can be viewed as both a framework and a toolkit: the framework can be used to build a model for the problem and the toolkit provides the components to build the solution algorithms. The first step, as a user, is to fill the framework with your own problem, the second is to use the toolkit to build your custom solution strategy. Once the first solving strategy has been set up the user can chose weather she is satisfied with the result or if she wants to personalise the algorithm even more, replacing some strategy with new ones that she can also provide herself.

1.2 Neighbourhood exploration based methods

Given a discrete optimization problems, the aim of the library is to allow easy switching between some of the most effective neighbourhood search algorithms: local search, random restart local search, variable neighborhood search, iterated local search, simulated annealing, tabu search and possibly others.

The exploration of a neighbourhood is at the root of all the implemented algorithms.

Given a discrete optimisation problem

$$\min_{s \in \overline{S}} \hat{f}(s) \tag{1.1}$$

$$\overline{S} \quad \text{is discrete} \tag{1.2}$$

the exploration can start from any point $s_0 \in \overline{S}$. Operators are defined as rules to modify the current point s_c into neighbouring points. The set of all the possible neighbouring points that can be reached is called *neighbourhood* of s_c and denoted by $\mathcal{N}(s_c)$. The algorithm can explore the neighbourhood completely or not depending on the neighbourhood exploration scheme chosen. A new current point s'_c is chosen from the neighbourhood based on an acceptance rule – the heart and soul of the algorithm – that depends, possibly amongst other things, on the cost function $\hat{f}$. If no point can be accepted the algorithms terminates.

The acceptance rule is the ingredient that is most characterising of the search algorithm:

Local Search the best point in the neighbourhood is accepted if it improves over the current point:

$$s'_c = argmin \left\{ s \in \mathcal{N}(s_c) : \hat{f}(s) \leq \hat{f}(s_n) \wedge \hat{f}(s) < \hat{f}(s_c) \quad \forall s_n \in \mathcal{N}(s_c) \right\} \tag{1.3}$$

Simulated Annealing uses a probabilistic rule in analogy with the physical annealing of metals. A point from the neighbourhood is accepted with probability

$$P = \max \left\{ 1, \exp \left(\frac{\hat{f}(s_c) - \hat{f}(s_n)}{T} \right) \right\} \tag{1.4}$$

where T is the "temperature" of the algorithm. The T parameter varies from some starting value, usually very high, to a very low value. Temperature is updated at each iteration of the algorithm based on a predefined cooling schedule.

Tabu Search in tabu search a new point is always accepted, but *tabu* points are excluded from the search.

$$s'_c = argmin \left\{ s \in \mathcal{N}(s_c) \backslash tabu(\mathcal{N}(s_c)) : \hat{f}(s) \leq \hat{f}(s_n) \quad \forall s_n \in \mathcal{N}(s_c) \right\} \tag{1.5}$$

except for the degenerate case in which $\mathcal{N}(s_c) \backslash tabu(\mathcal{N}(s_c)) = \emptyset$. The criterion that is used to declare a certain point of the neighbourhood as *tabu* is based on a short term memory and will be analysed further in the following paragraphs.

The local search is monotonic and will stop in a point of local minimum. This limitation can be mitigated by non monotonic search algorithms such as Simulated Annealing or Tabu Search that will explore the space aiming at a cost decrease on the long term but accepting also non-improving moves to escape local optima and implement specific strategies to avoid cycling.

Sometimes the explored set $\overline{S}$ set is not *feasible* in a strict sense and sprouts from the relaxation of an original feasible space S that is more difficult to explore. When this happens the objective function $\hat{f}$ is usually a combination of the real cost f with some terms that penalise the solutions $s \in \hat{S} \backslash S$. This tactic is very useful when, in the original feasible space S, the neighbourhood structure induces funnels between regions of the feasible space (or this regions are simply plain unreachable each other). The ability to explore a relaxed space, with less constraints, can ease moving from one region to another using an unfeasible path. Some algorithms try to follow the border between feasibility and infeasibility introducing a penalising term, proportional to the current infeasibility, using a proportion that is increased after each infeasible iteration and decreased after each feasible one.

1.2.1 Local search

The local search implemented in METS*lib* can be configured in two different ways: it can accept the best solution of the neighbourhood or the first improving solution that is found exploring the neighbourhood.

The search will stop when no improving move can be found in the neighbourhood.

Algorithm 1 Local search

$s^* \leftarrow s_0$
loop
  $s'' = s^*$
  for all $s' \in N(s'')$ **do**
    if $f(s') < f(s'')$ **then**
      $s'' = s'$
      { *we can stop the search on the best or on the first improving point* }
    end if
  end for
  if $f(s'') < f(s^*)$ **then**
    $s^* \leftarrow s''$
  else
    return s^*
  end if
end loop

1.2.2 Simulated annealing

The search by Simulated Annealing was proposed in [10]. Many variants exist in literature, the one proposed in the METS*lib* follows the general scheme given in algorithm 2 with an additional termination criterion and the ability to replace the cooling schedule with the desired one.

In the current state the Simulated Annealing procedure implementation is less customisable than the Tabu Seach we will discuss in section 1.2.3.

Algorithm 2 Simulated Annealing

$s'' \leftarrow s_0$
$s^* \leftarrow s_0$
$T \leftarrow T_0$
while $T > 0$ **do**
  for all $s' \in N(s'')$ **do**
    pick random u in U_{01}
    if $\max\{1, e^{\frac{f(s')-f(s'')}{T}}\} > u$ **then**
      {*always accept improving moves, accept also non improving moves with probability* $e^{\frac{f(s')-f(s'')}{T}}$}
      $s'' = s'$
      break { *stop exploring the current neighbourhood after accepting a point* }
    end if
  end for
  if $f(s'') < f(s^*)$ **then**
    $s^* \leftarrow s''$
  end if
  $T \leftarrow update(T)$
end while
return s^*

The acceptance rule, is loosely based on the transitions of a physical system during the process of annealing. It is easier and probably more useful to drop the analogy and simply state the rule as a probabilistic acceptance rule that allows deteriorating moves with a probability function that depends on the current iteration and is usually very high at the beginning and very low at the end of the algorithm. In this way the algorithm will proceed with a good breadth at the beginning and with a good intensification near the end of the algorithm.

It can be proved [9], working on the probability of the state of the algorithm and of the transitions that Simulated Annealing converges with high probability to a global optimum under some assumptions, most notably a finite problem, an exponential cooling, and an number of iterations that leads to infinity. It's probably worth noting that, for a finite combinatorial problem, explicit enumerations – a technique useful only for small problems with a search space, right now, not much bigger than 15! – gives similar guarantees. The most important

characteristic of Simulated Annealing is not the theoretical convergence but the well balanced non monotonic behaviour that leads to good solutions in a limited number of iterations.

Another obvious requirement for a well working implementation of a meta-heuristic algorithm is a non zero transition probability between any two points of the feasible space. This requires in first analysis a well thought neighbourhood structure.

In our implementation the probabilistic rule is hard-coded in the algorithm in the current version of METS*lib*. On the other hand the cooling schedule can be freely replaced and customised.

1.2.3 Tabu search

Tabu search [8, 7] is the main focus of the METS*lib* framework. It is the most tested and also, given the effort required to set it up, one of the most effective on a wide range of practical combinatorial problems.

In tabu search the "best" neighbouring solution is always accepted with maximum disregard of the current point and the value of its cost function.

The best solution is chosen as the one that has the best cost and has not been declared tabu by the search memory unless it meets some criterion of general aspiration. The tabu memory is frequently called *tabu list* and its length is called *tenure*.

The aspiration criterion is used to override the memory mechanism when it becomes too influential. In some circumstances the tabu memory can inhibit solutions that have bright characteristics: like being the best solution ever found, being feasible w.r.t. a relaxed constraint or being the best solution having some other distinctive attributes.

Contrariwise the tabu memory has the role of avoiding certain solution based on the search history of the algorithm in an attempt to avoid cycling. It can be implemented in many different ways and is a very important feature of the tabu search: it can inhibit visiting again a certain solution for a certain number of iterations (the *tenure* of the list) or it can memorise and compare only some attributes of the neighbouring solution, or of its difference to the current one, or based on some other strategy.

Tabu lists have been implemented in may ways: as FIFO (first-in first-out) queue, as LRU (least recently used) cache memories, as look-up tables, or in less ordinary ways.

In the knowledge of the author a very common type of tabu list is based on the memory of the last *tenure* moves. Each move is represented by a code, independent on the current point, and the same move is not done again for tenure iterations unless the aspiration criterion is met .

Algorithm 3 Tabu Search

$s'' \leftarrow s_0$
$s^* \leftarrow s_0$
while $\neg terminate()$ **do**
  $z_n = \infty$
  for all $s' \in N(s'')$ **do**
    if $f(s') < z_n \wedge (\neg tabu(s') \vee aspiration\_met(s'))$ **then**
      $s'' \leftarrow s'$
      $z_n \leftarrow f(s'')$
    end if
  end for
  if $z_n < f(s^*)$ **then**
    $s^* \leftarrow s''$
  end if
end while
return s^*

1.3 The curse of local search methods

Local search methods are generally condemned by their own locality. They can usually intensify the search very well in a certain zone, but have a hard time trying to get from one point to a very distant one.

For combinatorial problems the foremost method to avoid excessive locality is to design a neighbourhood structure that leads to a good balance between neighbourhood size and the overall diameter of the graph obtained connecting the points of the search space with their neighbours.

In some circumstances this is very hard to achieve or is not sufficient and some other methods are required to diversify the search.

Intensification and diversification are terms that are very common in the tabu search vocabulary and have been interpreted in many different ways. Usually a good algorithm design consists in diversification phases interspersed with an intensification of the search. How this is achieved is a matter of experience and trial and error. However there are some useful diversification methods that are utterly general.

Sometimes the diversification can be achieved with a longer or dynamic tabu list, other times using a penalisation on recurring situations (somewhat misusing the objective function).

In many situations we can resort to methods that can be used to improve the diversification of a metaheuristic in a general way. We will discuss in the following paragraph the two that are most common: multi-start and iterated local search.

1.3.1 Multi-start

Multi-start [7, 519-537] is a very simple and general diversification method. In order to better explore distance portions of the search space the search is started more that one time from different points, carefully chosen, and the best overall point is recorded.

Algorithm 4 Multi-start

```
i ← 1
s* ← ∅
while ¬terminate() do
  s_0 ← generate(i-th starting point)
  s' ← improve(s_0)
  if f(s') < f(s*) then
    s* ← s'
  end if
end while
return s*
```

More frequently than not, the "generate" function will provide a starting point using a so called Monte Carlo method [4]. Choosing a random starting point without any bias, with the added handicap of having to do so using a computer, is not a plain trivial task. Citing what D. Knuth wrote in *The Art of Computer Programming*: "[...]*random numbers should not be generated with a method chosen at random.* Some theory should be used.". In order to avoid any type of bias a well thought and well implemented algorithm is of uttermost importance.

In recent years [6] the multi-start approach has been merged with the willingness to make good use of the search history. The most prominent variation, in this direction, on the multi-start method is probably the Greedy Randomized Adaptive Search Procedure (GRASP) that is generally attributed to [2] and received a lot of attention in the latest years. In GRASP the starting solutions are typically generated with a randomized greedy procedure and improved with a local search, in the same way a multi-start does. As the algorithm proceeds the randomized starting solutions are generated with a bias guided by the similarity with previously generated points and the quality they achieved.

1.3.2 Iterated Local Search

Iterated Local Search (ILS) was introduced in [13, 12] although a very similar algorithm, introduced for continuous global optimization, and known as Basin-Hopping was introduced by [16].

ILS has a structure similar to multi-start, but instead of generating a new random solution at each iteration we apply a perturbation so that the new starting point has a good balance of similarity and dissimilarity from the previous solution.

Algorithm 5 Iterated local search

$s^* = \text{improve}(s_0)$
repeat
  $s' = \text{perturbation}(s^*)$
  $s'' = \text{improve}(s')$
  $s^* = \text{acceptance-criterion}(s'', s^*)$
until termination criteria met
return s^*

The perturbation can be applied to the best solution found during the search or to the last solution found by the improving procedure. In the first case we intensify the search around the best known solution while in the latter we diversify the search more. The usual choice will fall between always applying the perturbation to the best know solution or balancing the two methods to start with a better diversification and conclude with a better intensification.

1.3.3 Population based methods

Another common method to obtain a good diversification of the search is to maintain a population of elements and to obtain diversification exchanging information between elements (as happens in Evolutionary Algorithms), this is outside the scopes of this dissertation and of the metaheuristics library here presented.

METS*lib* does not offer, right now, any kind of support for population methods. However it can be certainly used to represent instances of the population and to improve those instances with local search methods when needed.

Chapter 2

The problem framework

2.1 Introduction

This chapter will be a more technical introduction to the METS*lib* framework that will require some knowledge of metaheuristics, of the C++ programming languages, and of some object orientation concepts. We will describe the classes of the modelling and of the solution framework and will propose some usage examples and results for the Quadratic Assignment Problem.

For a complete reference to the classes and methods of the library you should consult the complete API documentation available on-line [14].

2.2 The problem and neighbourhood framework

The first thing that the user needs to do in order to start solving a discrete optimisation problem using METS*lib* is to model the problem: in this process the user has different choices. The author's suggestion is to start implementing the simplest thing that could possibly work and to start improving over that working base.

On the other hand the choice of an efficient data structure depends deeply on the neighbourhood that we want to exploit, and will weight on the final computational time. So, before starting with the implementation a long consideration to guess the most appropriate neighbourhood and data structure is due.

METS*lib* facilitates the user mostly by providing standard and customary solvers. The solving strategy can be customised on a working code base, when the objective function and the neighbourhood have been tested deeply.

In some circumstances, when the problem we want to model is a permutation problem, the framework will facilitate the work even more, providing ready made operators and neighbourhoods to start with.

The components that are involved in the modelling are:

`mets::feasible_solution` - A generic tag for a solution

mets::evaluable_solution - A solution with an objective function that can be used with solver components that need to evaluate solutions.

mets::permutation_problem - A partially specialised solution to model the wide range of problems in which the solution space is made by all the possible permutations of a given set.

mets::move - A generic move that can be evaluated or applied against a solution instance.

mets::mana_move - A move that can be stored in the hash table of the mets::simple_tabu_list.

mets::swap_elements - A move that swaps two elements of a permutation problem.

mets::invert_subsequence - A 2-opt move (TSP like) on a permutation problem.

2.2.1 Feasible solutions

The role of the mets::feasible_solution class is to act as a base class for all problem instances. It is an empty class that you can fill with your problem data, variables and methods. The purpose for this tagging class is simply to have a common base class that can be used to pass the solution instances around.

In most cases the user will not derive directly from the empty feasible solution class, but will probably prefer the mets::evaluable_solution or some of its specialisations.

The mets::evaluable_solution base class has some pure virtual methods that must be overridden and allow to use the implemented solution with some toolkit components. The two virtual methods are the cost_function() and the copy_from(). This methods are used by most of the solution toolkit components to evaluate and store a solution in memory.

A generic evaluable solution instance represents a point in the search space and its objective value. Usually classes derived from this one will also provide helper methods to modify and manipulate the solution.

A special type of feasible solution specialisation is the mets::permutation_problem: a class that provides some facilities to explore a feasible space made of all the possible permutations of a given set. Very well known examples of permutation problems are the Travelling Salesman Problem, where the minimum cost permutation of cities has to be found and Assignment Problems where we have to assign facilities to locations. To some extent even the Vehicle Routing Problem that we will discuss in the next part can be modelled as a permutation problem.

2.2.2 Neighbourhood operators

Strictly speaking a move (also called an operator) is an operation that generates a single neighbour from a given point. There are two basic types the user can derive from: a generic mets::move and a tabu search specific mets::mana_move which

is, in turn, a generic move object with some methods that allows efficient storing and retrieving from a tabu list.

In METS*lib* moves provide two basic methods that are used by search algorithms:

`evaluate()` - evaluates the cost function of the provided solution instance after the move (without modifying the point)

`apply()` - modifies the solution into one of its neighbours.

The reason of having two distinct methods is that, in many situations, the evaluation of the neighbouring points can be done without destroying the starting point. When this is possible many costly copying operations can be avoided with a significant speed increase.

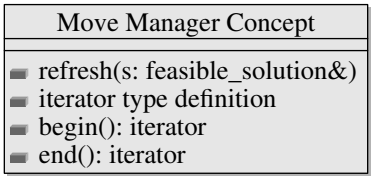
Another speed-up can be achieved implementing the cost function so that it does not recompute the whole objective function at each step, but updates the cost function considering only the variables that are being modified by the move. In the paragraph about Quadratic Assignment Problem we will illustrate how to improve the computation of the objective cost after a tentative move.

Typical examples of a move are: insert or remove a specific element from a certain position of a set, swap the position of two elements, increment or decrement a certain variable, and so on. Moves depend deeply on the data structure used: if a solution is represented as a vector the insert/remove/swap are natural moves, if the solution is represented as a tree it is easy also to move a sub-tree from one node to another. The specific move to be implemented depends deeply on the user insight on the problem.

2.2.3 Neighbourhood exploration

In METS*lib* move managers are containers of moves and as such represent the neighbourhood of the point to be explored. The neighbourhood plays a key role in the algorithm definition: the neighbourhood (and the resulting algorithm) can be deterministic or stochastic, define the search space allowing only feasible or also unfeasible points, they can be made using a single type of move or multiple types and so on.

Although the move manager exists as a framework class (see `mets::move_manager`, it can also be used as a generic concept [1]. There is no requirement to derive from an existing move manager. All the search algorithms are templetised and can use, as a neighbourhood, any class that will respect the following concept class:



At each iteration the search algorithm will call the `refresh()` method passing the current point to be explored.

All algorithms will thereafter explore the neighbourhood in order, using C++ instructions very similar to the following (assuming `move_manager` is an instance of a generic `move_manager_type` templetised class):

Listing 2.1: Neighbourhood expansion

```
1 move_manager.refresh(working_solution_m);
2 for(typename move_manager_type::iterator
3     move_it = move_manager.begin();
4     move_it != move_manager.end();
5     ++move_it)
6   ...
```

Your neighbourhood can reach different level of complexity.

Constant neighbourhood - if the neighbourhood does not depend on the point being considered, i.e. the moves does not depend on the current solution (like in *swap the third element with the seventh*), the neighbourhood can be any container with an empty *refresh* method. The `mets::move_manager` base class can be used in this case.

Neighbourhood depends on the current point - in this case you also need to implement the refresh method and fill the container at each iteration with the feasible moves.

Implicit exploration - an implicit exploration of the neighbourhood is doable with a specialised refresh method that only inserts the promising moves.

Customised neighbourhood exploration - when the `mets::move_manager` is no more sufficient (e.g. the generation of the neighbourhood at each iteration is too expensive) the user can implement her own iterators and neighbourhoods that adhere to the concept: in this case the moves can be generated by the iterator one at a time.

2.2.4 Modelling a permutation problem

If the problem that the user wants to model is a permutation problem, a problem in which the solution space can be represented as all the permutation of a certain set, the `mets::permutation_problem` base class can be used. Deriving from the permutation problem class the user can employ the provided neighbourhoods and operators. The variable will be called `pi_m` and, upon construction, will be a vector (an ordered set) containing the first n natural numbers.

The user needs to implement the pure virtual methods `evaluate_swap()` and `compute_cost()`, and, in the likely case where new variables and parameters have been introduced, also the `copy_from()` method.

The provided neighbourhoods are:

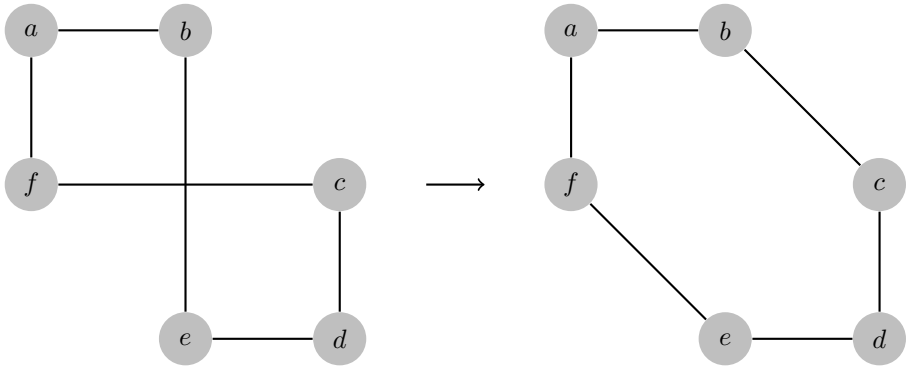


Figure 2.1: A 2-opt exchange move on an Hamiltonian cycle consists in removing two arcs and restoring the cycle in a different way (note that there is only one way to do this). After the exchange the sequence changes from $abedcf$ to $abcdef$. Exchanging arcs (b, e) and (c, f) with (b, c), (e, f) is equivalent to inverting the subsequence contained between b and f. We are not aware of other works where this particular data structure has been used.

`mets::swap_full_neighbourhood` - a neighbourhood made from all the possible swaps between every two elements of the permutation problem.

`mets::swap_neighbourhood` - a random subset uniformly chosen from the previous one.

`mets::invert_full_neighborhood` - a 2-opt neighbourhood made inverting all the subsequences of the current permutation (see fig. 2.1).

Chapter 3

The solution toolkit

3.1 Introduction

Once the user has filled the framework with her problem, she can apply one of the several algorithms provided by the library. Some of them are almost ready to use, others require some coding. As the library improves the coding needed will be reduced. The default direction of the optimisation is minimisation: in order to maximise the sign of the cost function must be inverted.

The search algorithms are modular. Each algorithm, before starting the search, must be configured using the required components.

First we must provide the search with a starting point[1] and a solution recorder[2] . The former is needed to give the search the s_0 solution to start the search from, while the latter is used to record the best solution found during the search.

Some algorithms can use an external stopping criterion while others will need algorithm specific components. Tabu search will need a tabu list and an aspiration criterion, while simulated annealing will need a cooling schedule.

Each component can be interchanged independently with other provided components or with components developed by the user.

3.1.1 Recording the best solution

The framework provides a `mets::best_ever_solution` recorder that can be used together with `mets::evaluable_solution` instances to record the solution with the best cost. This recorder will copy improving points into the instance stored into the recorder via the `accept()` method.

Sometimes the merit function value is not sufficient to accept a point as improving. A widespread example of this situation, as we mentioned in 1.2, is when the feasible space being explored does not match the feasible space for

[1] An instance of a class derived from `mets::feasible_solution`

[2] A class derived from `mets::solution_recorder`

the original problem. In this cases we may want to implement our own recorder that will accept a solution as improving if it is the best known solution that is also feasible for the un-relaxed model.

3.1.2 Terminating the search

Tabu Search and Simulated Annealing algorithms can be terminated by an external termination criterion. This is useful to stop the search altogether or to restart the search, possibly with different parameters or a different algorithm.

The termination criteria provided are:

`mets::iteration_termination_criterion` - The search is stopped after a fixed number of iterations.

`mets::noimprove_termination_criterion` - The search is stopped after a fixed number of non-improving iterations. This stopping criterion provides a `second_guess()` method that will return the maximum number of non improving iterations that was below the threshold. This value can be used to have some hints on how much iterations have been wasted.

`mets::threshold_termination_criterion` - The search is stopped when the objective function drops below a certain value.

`mets::forever` - The search is never stopped.

Termination criteria can be chained as in the chain of responsibility design pattern [5]. When two or more criteria are chained the search will stop when any termination criterion is met.

3.2 Implemented algorithm

An interesting feature of METS*lib* is that it is extremely easy to start the search using a local search, then switch to SA or TS (or even try the two together) and react during the search making decisions that depends on the particular solution being explored.

All search algorithms derive from the `mets::abstract_search` templetised class. The class accepts a neighbourhood exploration type as described in 2.2.3. It is also an observable subject as in the observer design pattern [5].

Observers obeying the `mets::search_listener` interface can use `attach()` to be notified (we can also say called back) via the `update()` method. Observers are notified on different events: when a move is made, when an improvement occurs or in other circumstances that can be detected using the `step()` method of the search algorithm. As an example, the following code can be used to observe any search algorithm using an evaluable solution to report improvements on a stream:

Listing 3.1: A simple search listener

```
template<typename neighborhood_type>
struct logger : public mets::search_listener<neighborhood_type>
{
  logger(std::ostream& o) :
    mets::search_listener<neighborhood_type>(),
    iteration(0), os(o) { }

  void update(mets::abstract_search<neighborhood_type>* as) {
    mets::evaluable_solution& e = dynamic_cast<mets::
        evaluable_solution&>(as->working());
    if(as->step() == mets::abstract_search<neighborhood_type>::
        MOVE_MADE)
        iteration++;
    else if(as->step() == mets::abstract_search<neighborhood_type>::
        IMPROVEMENT_MADE)
        os << iteration << e.cost_function() << '\n';
  }
  int iteration;
  std::ostream& os;
};
```

The only method, excluding the constructor is `update()` that is called upon events by the framework and receives the current algorithm. The implemented update procedure checks the value of the algorithm `step()`, documented in the API documentation, and prints information about the current iteration only after an improving move has been made.

3.2.1 Local search

The simplest available algorithm is hill climbing, where the neighbourhood is explored to search for the first improving move (in the minimising direction). A simple variation is the Steepest descent, where the neighbourhood is fully explored to find the best improving one. Both algorithms terminate when there are no more improving moves in the neighbourhood.

This strategies are implemented in the `mets::local_search` class.

3.2.2 Random Restart Local Search

The Random Restart Local Search (RRLS) is probably not the most effective metaheuristics, but can be used for comparisons to test if another metaheuristic performs well on a given problem. The reason is that it is really simple and relies only on uniform randomness and a neighbourhood structure. It consists of a multi-start method combined with a local search.. To be fair, when comparing with other algorithms, comparison should be made using an uniform random generation and the same neighbourhood structure and exploration.

The generation of the starting point depends on the problem and can be done using the TR1 random extensions to the C++ standard (or using some external library the user is already familiar with). The RRLS is not provided

by the framework but is really simple to implement using a single `for` cycle and `mets::local_search`.

3.2.3 Simulated Annealing

The algorithm described in 1.2.2 is implemented in `mets::simulated_annealing` together with two basic cooling schedules: linear cooling schedule [15] defined in `mets::linear_cooling`, and exponential schedule [10] `mets::exponential_cooling`.

The METS*lib* implementation will stop when the temperature reaches T_{min} or when an external termination criterion is met. If the termination criterion is satisfied first may want to check that the temperature reached is not too high, do something with the solution, and in case start another search phase.

3.2.4 Tabu Search

Tabu search (see 1.2.3) is implemented in the `mets::tabu_search` class. The search algorithm must be built providing the starting point, the solution recorder, the neighbourhood exploration strategy, a tabu list, an aspiration criterion, and a termination criterion.

The starting point (2.2.1), the solution recorder (3.1.1), the neighbourhood ((2.2.2,2.2.3) have been discussed earlier in this chapter.

The tabu list must be an instance of a class derived from `mets::tabu_list_chain`. The ready to user `mets::simple_tabu_list` is a tabu list that memorises the latest *tenure* moves and will avoid to explore them if present in the current neighbourhood. Lots of other strategies are possible and can be easily implemented. Tabu lists can be chained together: a neighbourhood element will be judged tabu if any of the tabu lists matches.

The last needed component, the best ever aspiration criterion, is implemented in `mets::best_ever_criterion`.

3.3 An application

We've seen how METS*lib* framework can be used to model discrete and combinatorial optimisation problems through the definition of a feasible space to be explored, operators, and neighbourhoods.

The modelled problem can than be used with any of the available algorithms. Algorithms are, in turn, modular and can be built from replaceable first principles components.

In the next section we will see actual code to model the Quadratic Assignment Problem, and different method to solve it using increasingly sophisticated algorithms.

3.3.1 The Quadratic Assignment Problem

The Quadratic assignment problem is usually attributed to Koopmans-Beckmann (1957) [11, 3].

In the Quadratic Assignment Problem we are given a set of n locations and n facilities and we need do decide where to build the facilities (assign facilities to locations) so that the cost of inter-plant transportation is minimised.

We call $F \in \mathbb{R}^{n \times n}$ the matrix of the flows f_{ij}, the number of commodities that we need to transport from facility i to facility j in a certain amount of time.

We call $D \in \mathbb{R}^{n \times n}$ the matrix of the costs d_{ij} of transporting one commodity from location i to location j.

A first formulation can be written introducing a permutation matrix X where the element x_{ij} is equal to 1 if the facility i is assigned to location j and zero in the other case.

The model can be written as follow:

$$\min \sum_i \sum_j \sum_p \sum_q f_{ij} d_{pq} x_{ip} x_{jq} \tag{3.1}$$

$$\text{subject to} \sum_j x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\} \tag{3.2}$$

$$\sum_i x_{ij} = 1 \quad \forall j \in \{1, 2, \ldots, n\} \tag{3.3}$$

$$x_{ij} \text{ binary} \tag{3.4}$$

This formulation can be misleading for an inexperienced operations research analyst in search for a useful neighbourhood. It may be useful for exact methods, but can lead to neighbourhoods that are hard to explore.

If we define Π as the set of all the possible permutations of the set $\{1, 2, \ldots, n\}$ and $\pi \in Pi$ as a particular permutation so that π_i will represent the position chosen for the i-th facility we can restate the problem using another well known formulation, that is more natural from the point of view of a neighbourhood exploration:

$$\min_{\pi \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)}. \tag{3.5}$$

In this formulation the neighbourhood operators emerge almost on their own: the algorithm can move from a permutation to another by swapping elements.

The full problem definition can be written in a very compact and efficient form using METS*lib*:

Listing 3.2: QAP model

```
1 class qap_model : public mets::permutation_problem
2 {
3 protected:
4   std::vector< std::vector<int> > f_m;
5   std::vector< std::vector<int> > d_m;
6
```

```
7   double compute_cost() const {
8     double sum = 0.0;
9     for(unsigned int ii = 0; ii != pi_m.size(); ++ii)
10      for(unsigned int jj = 0; jj != pi_m.size(); ++jj)
11        sum += f_m[ii][jj] * d_m[pi_m[ii]][pi_m[jj]];
12    return sum;
13  }
14
15 public:
16  qap_model() : permutation_problem(0), f_m(), d_m() {};
17  void copy_from(const mets::copyable& sol) {
18    const qap_model& o = dynamic_cast<const qap_model&>(sol);
19    permutation_problem::copy_from(sol);
20    f_m = o.f_m;
21    d_m = o.d_m;
22  }
23
24  double evaluate_swap(int i, int j) const {
25    double delta = 0.0;
26    for(unsigned int ii=0; ii != f_m.size(); ++ii) {
27        delta -= f_m[i][ii] * d_m[pi_m[i]][pi_m[ii]];
28        delta -= f_m[ii][i] * d_m[pi_m[ii]][pi_m[i]];
29        delta -= f_m[j][ii] * d_m[pi_m[j]][pi_m[ii]];
30        delta -= f_m[ii][j] * d_m[pi_m[ii]][pi_m[j]];
31        int ni = ii;
32        if(ii==i) ni = j; else if(ii==j) ni = i;
33        delta += f_m[i][ii] * d_m[pi_m[j]][pi_m[ni]];
34        delta += f_m[ii][i] * d_m[pi_m[ni]][pi_m[j]];
35        delta += f_m[j][ii] * d_m[pi_m[i]][pi_m[ni]];
36        delta += f_m[ii][j] * d_m[pi_m[ni]][pi_m[i]];
37    }
38    return delta;
39  }
40 };
```

The class starts declaring the two matrices `f_m` and `d_m` and uses the implicit `pi_m` permutation vector inherited from the permutation problem base class. The copy method at line 17 is used by the best ever solution recorder to store the best known solution, the compute method at line 7 is the C++ implementation of eq. (3.5) and computes the cost function form scratch. The most important method, evaluate swap at line 24, is used to evaluate the cost function after a swap using only $8n$ multiplications instead of the n^2 needed by a full cost computation.

3.3.2 Local search

In this paragraph the model declared in the previous one is used to implement a simple local search.

Listing 3.3: Local search strategy

```
1 qap_model problem_instance;
2 std::cin >> problem_instance;
3 unsigned int N = problem_instance.size();
```

```
4
5  qap_model best_solution(problem_instance);
6  mets::best_ever_solution best_recorder(best_solution);
7
8  mets::swap_full_neighborhood neighborhood(N);
9
10 std::tr1::mt19937 rng(time(NULL));
11 mets::random_shuffle(problem_instance, rng);
12
13 mets::local_search<mets::swap_full_neighborhood>
14       algorithm(problem_instance,
15                 incumbent_recorder,
16                 neighborhood);
17
18 mets::iteration_logger g(clog);
19 algorithm.attach(g);
20 algorithm.search();
21
22 std::cout
23     << std::fixed
24     << N << " "
25     <<  best_solution.cost_function() << std::endl
26     << best_solution << std::endl;
```

Lines 1–2 declare and load the working QAP instance, lines 5–6 declare a second solution instance to store the best known solution (this will closely follow the working instance for local search, but will be more useful for other algorithms). The neighbourhood is instantiated at line 8 and will explore all the possible single swaps. A random starting point is generated using the C++ TR1 extension and the random shuffle algorithm (lines 10–11): the algorithm will select a random permutation uniformly chosen between all possible permutations. The algorithm declaration (local search in this case) is at line 13. The following lines are used to attach a logger to the search (lines 18–19), to start the search and to print the result.

The complete code can be found on-line on the METS*lib* page of the COIN-OR project.

3.3.3 A simple Tabu Search

If we replace the statement at line 13 with the following code we can replace the local search with a tabu search algorithm. There is no need to change the model or the other solver statements.

Listing 3.4: Tabu search strategy

```
1 mets::simple_tabu_list tabu_list(N*sqrt(N));
2 mets::best_ever_criteria aspiration_criteria;
3 mets::noimprove_termination_criteria termination_criteria(1000);
4 mets::tabu_search<swap_full_neighborhood_t>
5     algorithm(problem_instance,
6               incumbent_recorder,
7               neighborhood,
8               tabu_list,
```

```
9               aspiration_criteria,
10              termination_criteria);
```

This code declares the needed components and builds a simple tabu search strategy.

3.3.4 Random Restart Local search

In order to random restart the aforementioned local search algorithm shown in listing 3.3 the termination and recording component of METS*lib* can be used in a loop containing the local search as show in listing 3.5.

Listing 3.5: RRLS

```
1  mets::iteration_termination_criteria rrls_stop(200);
2  while(!rrls_stop(best_recorder.best_seen())) {
3    mets::random_shuffle(problem_instance, rng);
4    mets::local_search<mets::swap_full_neighborhood>
5          algorithm(problem_instance,
6                    best_recorder,
7                    neighborhood);
8    mets::iteration_logger g(clog);
9    algorithm.attach(g);
10   algorithm.search();
11 }
```

3.3.5 An Iterated Tabu Search

We also tested an Iterated Tabu Search (ITS), replacing the strategy at line 13 of listing 3.3 with the following strategy that uses a randomly varying tabu list tenure, and perturbation entity. In this algorithm the neighbourhood has been replaced by a random sample of neighbours of size $N\sqrt{N}$.

Listing 3.6: ITS

```
1  // random number generators
2  std::tr1::mt19937 rng(time(NULL));
3  std::tr1::uniform_int<int> tenure_gen(5, N*sqrt(N));
4  std::tr1::uniform_int<int> pert_gen(N/5, N/2);
5  mets::random_shuffle(problem_instance, rng);
6  mets::noimprove_termination_criteria ils_stop(20);
7  mets::simple_tabu_list tabu_list(tenure_gen(rng));
8  while(!ils_stop(ils_recorder.best_seen())) {
9    qap_model ts_solution(problem_instance);
10   mets::best_ever_solution ts_recorder(ts_solution);
11   mets::best_ever_criteria aspiration_criteria;
12   mets::noimprove_termination_criteria ts_stop(500);
13   mets::tabu_search<swap_neighborhood_t>
14     algorithm(problem_instance,
15               ts_recorder,
16               neighborhood,
17               tabu_list,
18               aspiration_criteria,
```

Table 3.1: Comparison of results on Taillard instances for the presented algorithms.

		LS gaps (%)		RRLS gaps (%)		TS gaps (%)		ITS gaps (%)	
Instance	B.K.S.	Avg	Min	Avg	Min	Avg	Min	Avg	Min
tai12a	224416	10.20	6.23	0.00	0.00	0.49	0.00	0.00	0.00
tai15a	388214	2.19	2.34	0.27	0.00	0.71	0.00	0.00	0.00
tai17a	491812	5.38	3.10	1.34	0.00	1.43	0.56	0.13	0.00
tai20a	703482	6.00	2.33	1.55	0.47	1.63	0.47	0.75	0.00
tai25a	1167256	5.09	3.58	2.21	1.88	2.46	1.78	0.98	0.00
tai30a	1818146	4.69	2.33	2.36	1.84	2.11	1.39	1.03	0.49
tai35a	2422002	5.25	3.95	2.79	2.28	2.11	1.42	1.47	0.81
tai40a	3139370	4.98	3.63	2.99	2.62	2.29	1.79	1.40	0.95
tai50a	4938796	5.04	4.13	3.16	2.10	2.54	2.10	2.07	1.75
tai60a	7205962	4.67	3.98	3.32	3.00	2.31	2.04	1.97	1.47
tai80a	13511780	4.22	3.67	2.95	2.71	2.14	1.93	2.10	1.69
tai100a	21052466	3.60	3.13	2.81	2.65	1.98	1.77	2.00	1.82
Avg		5.11	3.53	2.15	1.63	1.85	1.27	1.16	0.75

```
19                ts_criteria);
20    algorithm.search();
21    ils_recorder.accept(ts_recorder.best_seen());
22    problem_instance.copy_from(ils_recorder.best_seen());
23    mets::perturbate(problem_instance, pert_gen(rng), rng);
24    tabu_list.tenure(tenure_gen(rng));
25  }
```

3.4 Computational results

We tested the implemented strategies on the Taillard instances and compared with the best known solutions (BKS) of the QAPLIB (Burkard, ela, Karisch and Rendl) solving each instance 10 times and providing average and minimum gaps. The complete set of algorithms presented is provided with full source in the *Examples* package, available for download on `www.coin-org.org` . The results of ITS, on average below 1% can be considered very good in practice as they are far less than the tolerance in road measurement.

Chapter 4

Conclusions

We presented METS*lib* a metaheuristics framework for OR researchers and practitioners searching for a simple, efficient, reusable and concise framework for local search methods. We've shown the framework spirit and main components giving sample code where needed. We also provided some results on some very simple code using only stock components, that, in the author opinion, are very satisfying. Starting from this tutorial the user can deepen her knowledge of the framework reading the example code and the almost complete API documentation, both available on-line.

METS*lib* is still young. The current release, version 0.5.2, can be downloaded from `http://www.coin-or.org/metslib`. Software of this kind is not of widespread use and untested code lies in ambush. As a Free Software user, on one hand you have the right to demand for reliable and stable code, on the other hand you should test code, report bugs, join the mailing list, encourage the authors, tell about framework usages, and, in general, find your way to participate in its development.

Index

Bibliography

[1] Gabriel Dos Reis and Bjarne Stroustrup, *Specifying c++ concepts*, SIGPLAN Not. **41** (2006), 295–308.

[2] Thomas A. Feo and Mauricio G. C. Resende, *A probabilistic heuristic for a computationally difficult set covering problem*, Operations Research Letters **8** (1989), no. 2, 67 – 71.

[3] Gerd Finke, Rainer E. Burkard, and Franz Rendl, *Quadratic assignment problems*, Surveys in Combinatorial Optimization (Michel Minoux Silvano Martello, Gilbert Laporte and Celso Ribeiro, eds.), North-Holland Mathematics Studies, vol. 132, North-Holland, 1987, pp. 61 – 82.

[4] George S. Fishman, *Monte carlo: concepts, algorithms, and applications*, Springer, New York, 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[6] M. Gendreau and J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., International Series in Operations Research and Management Science, vol. 146, Springer, 2010.

[7] Fred W. Glover and Gary A. Kochenberger (eds.), *Handbook of Metaheuristics*, International Series in Operations Research and Management Science, vol. 114, Springer, January 2003.

[8] Fred W. Glover and Manuel Laguna, *Tabu Search*, 1 ed., Springer, June 1998.

[9] V. Granville, M. Krivánek, and J. P. Rasson, *Simulated annealing: A proof of convergence*, IEEE Trans. Pattern Anal. Mach. Intell. **16** (1994), 652–656.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science, Number 4598, 13 May 1983 **220, 4598** (1983), 671–680.

[11] Eugene L. Lawler, *The quadratic assignment problem*, Management Science **9** (1963), no. 4, pp. 586–599 (English).

[12] Helena Lourenço, Olivier Martin, and Thomas Stülze, *Handbook of metaheuristics*, International Series in Operations Research and Management Science, vol. 57, ch. 11, Iterated Local Search, pp. 321–353, Kluwer Academic Publisher, 2003.

[13] H.R. Loureno, O. Martin, and T. Sttzle, *A beginners introduction to iterated local search*, Proceeding of the 4th Metaheuristics International Conference, 2001, pp. 1–11.

[14] Mirko Maischberger, *METSlib, a metaheuristics framework in C++*, http://projects.coin-or.org/metslib, July 2010.

[15] R. E. Randelman and Gary S. Grest, *N-city traveling salesman problem: Optimization by simulated annealings*, Journal of Statistical Physics **45** (1986), no. 5, 885–890, The problem of finding the shortest closed path connectingN randomly chosen points is one of the classicNp-complete problems. We show that the length of tour depends logarithmically on the cooling rateQ in a simulated Monte Carlo anneal. We speculate that this is a general property of allNp-complete problems.

[16] David J. Wales and Jonathan P. K. Doye, *Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms*, Journal of Physical Chemistry A **101** (1997), 5111–5116.