Chapter 1

FROM THEORY TO IMPLEMENTATION: APPLYING METAHEURISTICS

An Object Oriented Programming approach

I.J. García del Amo, F. García López, M. García Torres, B. Melián Batista, J.A. Moreno Pérez and J.M. Moreno Vega

Departamento de Estadística, I.O. y Computación
Universidad de La Laguna, 38271 La Laguna, SPAIN
{igdelamo,fcgarcia,mgarciat,mbmelian,jamoreno,jmmoreno}@ull.es

1. Introduction

Metaheuristics are strategies to design heuristic procedures. Since the first time the word *metaheuristic* appeared in the seminal paper of Tabu Search by Fred Glover in 1986 [8], there have been a lot of papers, reviews and books on Metaheuristics (see [29], [37], [31], [2], [12]). The classification of metaheuristics is usually based on the kind of procedures for which they are designed. For example, there are constructive metaheuristics like GRASP [30], evolutive metaheuristics like Genetic Algorithms [28] or neighborhood metaheuristics like the classical greedy local search. However, other possible classifications of metaheuristics are given by the computational tool or technique considered fundamental for the procedure, like Neural Networks [27] or Ant Colony Systems [3]. Some of the proposed algorithms are designed following not only one metaheuristic, but several of them. Moreover, some proposed metaheuristics are formed by mixed strategies that combine different kinds of tools, in such a way that they are either hybrid metaheuristics obtained by a combination of them, or they can be considered a simplified version of them obtained by ignoring some of the tools applied. It is also usual for many metaheuristic to have some modifications or adaptations to special circumstances that have been proposed to provide different versions or extensions of the metaheuristics. For these reasons the field of metaheuristics is continuously growing with new proposals that are

becoming efficient and effective for an increasing number of difficult optimization problems.

However, the most relevant classification of them is to separate the metaheuristics based in populations of solutions from the metaheuristics based on a single solution. Among the single solution based metaheuristics (or point-based solutions, as we will refer to from now on), we should emphasize the importance of some of them, like the Greedy Search, the Random Search, the Local Search [38], the Guided Local Search [33], the Simulated Annealing [22], the Tabu Search [13] or the Variable Neighborhood Search [21]. On the other hand, among the population based metaheuristics, some of the most important ones are the Ant Colony Systems [3], the Scatter Search [24], the Estimation of Distribution Algorithms [25] or the Genetic Algorithms [28]. Other classifications appear in [32] and [39].

1.1 Variable Neighborhood Search

Variable Neighborhood Search (VNS) [17], [18], [20], [19], [21] is a recent and effective metaheuristic for solving combinatorial and global optimization problems that is capable of escaping from the local optima by systematically changing the neighborhood structures within the search. VNS proceeds using a descent method to reach a local minimum, then explores, either systematically or at random, increasing neighborhoods of this solution. In each iteration, one or several points within the current neighborhood are used as an initial solution for a local descent. The procedure jumps from the current solution to a new one if and only if a better solution has been found.

It has been empirically observed that, for many problems, local minima with respect to one or several neighborhoods are relatively close to each other. Therefore, a local optimum often provides some information about the global one. This may for instance be several variables with the same value, but usually it is unknown which ones are such. An organized study of the neighborhood of this local optimum is applied, until a better one is found.

Variable Neighborhood Descent (VND) is a deterministic version of VNS based on the fact that a local minimum with respect to one neighborhood structure is not necessary so for another. Thus, each time the descent reaches a local optimum for a set of moves, the method changes the set of moves; it changes the neighborhood each time it is trapped by the neighborhood structure. The method thus takes advantage from combining several descent heuristics and it stops at a local minimum with respect to all the neighborhood structure. Since a global minimum

is a local minimum with respect to all possible neighborhood structures, the probability of reaching to the global optimum increases.

Another simple application of the VNS principle is the Reduced Variable Neighborhood Search (RVNS). From an initial solution, a point is chosen at random in the first neighborhood. If its value is better than the current one, the search is re-centered there. Otherwise, the search proceeds to the next neighborhood. After all neighborhoods have been considered, it starts again with the first, until a stopping condition is satisfied. Usual stopping criteria are based on a maximum computing time since the last improvement, or a maximum number of iterations.

In the previous two methods, we see how to use variable neighborhoods to descent to a local optimum and to find promising regions for near-optimal solutions. Merging the tools for both tasks leads to the General Variable Neighborhood Search scheme that uses VND to improve each solution sampled by RVNS. However, the basic VNS scheme (Figure 1.1) is obtained by combining a local search with systematic changes of neighborhoods around the local optimum found.

BVNS method

1 Find an initial solution x; choose a stopping condition;

2 Repeat until the stopping condition is met:
(1) Set $k \leftarrow 1$;
(2) Repeat the following steps until $k = k_{max}$:

(a) *Shaking.* Generate a point x' at random from the k^{th} neighborhood of x ($x' \in \mathcal{N}_k(x)$);

(b) *Local search.* Apply some local search method with x' as initial solution; denote with x'' the so obtained local optimum;

(c) *Move or not.* If the local optimum x'' is better than the incumbent x, move there ($x \leftarrow x''$), and continue the search with $\mathcal{N}_1$ ($k \leftarrow 1$); otherwise, set $k \leftarrow k + 1$;

Figure 1.1. Basic Variable Neighborhood Search Method

1.2 Scatter Search

Scatter Search (SS) [10], [11], [14], [15], [24] is an evolutionary algorithm that combines good solutions from a reference set ($RefSet$) to construct new ones exploiting the knowledge of the problem at hand. Genetic Algorithms [28] are also evolutionary algorithms in which a population of solutions evolves by using the mutation and crossover operators. These operators have a significant reliance on randomization to

create new solutions. Unlike the population in Genetic Algorithms, the $RefSet$ of solutions in Scatter Search is relatively small.

The principles of the Scatter Search metaheuristic were first introduced in the 1970s as an extension of formulations for combining decision rules and problem constraints. This initial proposal generates solutions considering characteristics of several parts of the solution space [7]. Scatter Search has an implicit form of memory, which can be considered as an inheritance memory, since it keeps track of the best solutions found during the search, and selects their good features to create new solutions. The Scatter Search Template, proposed by Glover in 1998 [10], summarizes the general description of Scatter Search given in [9].

Scatter Search consists of five components processes: *Diversification Generation Method*, that generates a set of diverse solutions, *Improvement Method*, that improves a solution to reach a better solution, *Reference Set Update Method*, which builds and updates the reference set consisting of $RefSetSize$ good solutions, *Subset Generation Method*, to produce subsets of solutions of the reference set, and *Solution Combination Method*, that combines the solutions in the produced subsets. A comprehensive description of the elements of Scatter Search can be found in [10], [11], [14], [15], [24] and [24].

The basic Scatter Search procedure (see Figure 1.2) starts generating a large set of diverse solutions Pop, which is obtained using the *Diversification Generation Method*. This procedure creates the initial population (Pop),which must be a wide set consisting of diverse and good solutions. Several strategies can be applied to get a population with these properties. The solutions to be included in the population can be created, for instance, by using a random procedure to achieve a certain level of diversity. An *Improvement Method* is applied to each solution obtained by the previous method reaching a better solution, which is added to Pop.

A set of good representative solutions of the population is chosen to generate the reference set ($RefSet$). The good solutions are not limited to those with the best objective function values. The considered reference set consists of $RefSetSize1$ solutions with the best objective function values and $RefSetSize2$ diverse solutions. Then $RefSetSize = RefSetSize1 + RefSetSize2$. The reference set is generated by selecting first the $RefSetSize1$ best solutions in the population and secondly adding $RefSetSize2$ times the most diverse solution in the population.

Several subsets of solutions from the $RefSet$ are then selected by the *Subset Generation Method*. The *Solution Combination Method* combines the solutions in each subset using their good features. Then, the *Improvement Method* is applied to the result of the combination to get an

improved solution. Finally, the *Reference Set Update Method* uses the obtained solution to update the reference set.

```
procedure Scatter Search
begin
    Create Population;
    Generate Reference Set;
    repeat
        repeat
            Subset Generation Method;
            Solution Combination Method;
            Improvement Method;
        until (StoppingCriterion1);
        Reference Set Update Method;
    until (StoppingCriterion2);
end.
```

Figure 1.2. Scatter Search Metaheuristic Pseudocode

1.3 From Theory to Practice

After this review of current *state-of-the-art* on metaheuristics, it is time to ask the main question this chapter aims to answer. How can we implement these metaheuristics? Moreover, is there a way of programming them such that we could implement more metaheuristics than those seen here, without having to code them all from the beginning? Well, we believe that the answer is "yes". But before getting into the work, we need to comment first on some necessary concepts, for instance, Object Oriented Programming (OOP).

Although it is not the purpose of this chapter to explain OOP paradigm and all of its features, we shall provide a brief description of it to contextualize the reading. Interested readers should refer to [1] for more information about OOP standards and features.

The OOP paradigm is a relatively new one. Although the first programs following its guidelines were developed in later 1960's, it has not been until 1990's that they had become widely spread. Traditional programming deals with functions (code) and variables (data), emphasizing the difference between them. We could say that traditional programming is a "code-based" approach, as programmers should rely on the efficiency and correctness of the code to resolve the task it was programmed to. OOP, on the other hand, focuses on dealing with objects, which are functionality units containing both data and code to manage it. This provides OOP with desirable properties such as modularity, encapsulation, abstraction, polymorphism or inheritance. We can say that

OOP is a "design-based" paradigm, as it relies on the architecture of the program and the objects that live and interact in it, considering the code of the objects virtually irrelevant. The challenge in OOP therefore is of designing a sane object system ([35]).

Considering the problem of designing methods to implement several metaheuristics and making them extensible and easy to code, we can naturally think of OOP as an appropriate candidate for doing it.

- We want to code common parts just one time, so we do not have to program all the metaheuristic again every time we want to use a new one. This can be done by encapsulating common code into superclasses, and creating subclasses which inherit from them for metaheuristic-dependant code. This also has the advantage that if we detect an error in a part of the code of a superclass, we only have to correct it once, and all the subclasses will be updated.

- We want the code to be easily extensible. As we mentioned earlier, what really matters in OOP is the way in which objects relate to each other (i.e., WHAT they do), and not the specific code they use to do it (i.e., HOW they do it). For example, if we have a *problem* object, and a *metaheuristic* object, we want the *metaheuristic* object to get the *problem* object and to produce a *solution* object. We only care about what are the specifications of the *metaheuristic*. If the requirements are met, we should not care about how the *metaheuristic* object gets the solution. So, in theory, it does not matter if the *metaheuristic* object uses the VNS search or the Scatter Search, as long as it produces a valid solution for the problem. Thus, to extend the classes to include a new metaheuristic, we only have to create a subclass of the appropriate class and reimplement the necessary methods to use the new algorithm. We are changing the insides, but for an external viewer, it will still remain as a *metaheuristic* object.

2. Class Hierarchy

In this section we will explain in detail each of the classes that form the hierarchy we propose (Figures 1.3, 1.4 and 1.5). We will start talking about the classes that define the structure of a problem and a solution (classes *Problem* and *Solution* respectively). Next, we will explain the general properties and methods we consider every metaheuristic should have (conforming the base class *Metaheuristic*). Then, we will continue with an explanation of why metaheuristics should be separated depending on if they are population-based or point-based. Finally, we

will end with the specific details of two examples of metaheuristics already reviewed in the previous section: a point-based one (VNS) and a population-based one (Scatter Search). After this, we will comment the class *StopCriterion* and its relationship with metaheuristic classes. Note that, although classes *MhT_VNS* and *MhT_SS* are represented in Figure 1.4, they inherit from classes *PointBased* and *PopulationBased* respectively, which are represented in Figure 1.3.

The explanation of each class will be preceded by an enumeration of its attributes and methods, along with a short, general description of it and some relevant comments that we think are useful to understand the purpose of the class.

In the enumeration section, we will present firstly the attributes of the class, and then, its methods. Every attribute and method is preceded by one of these three symbols:

- "+": expresses that the attribute/method is public.
- "−": expresses that the attribute/method is private of the class.
- "#": expresses that the attribute/method is protected (that is, only the class or some subclass of it can access it).

Apart from these symbols, if a method or class has its name written in *italics*, that means that the method or class is abstract, and therefore it has to be redefined by a subclass (if it is an abstract method) or a subclass needs to be created for an object to be instantiated (if it is an abstract class).

Every attribute/method will end by a colon followed by the type of the attribute/method. For example, " : bool" means that the attribute is of type *bool* or that the method returns a *bool* value.

For convention, we will consider that the accessor methods of every class (i.e., *get* and *set*) return copies of the attribute (if it is a *getter*) or make a copy of the parameter before assigning it to the attribute (if it is a *setter*). This assumption is for preserving data encapsulation and integrity, so that every method can safely work with the object's data without interfering with other methods. Nevertheless, we understand that, in some cases, working with copies can be simply unaffordable (for example, in problems in which a solution is formed by a high number of elements). In these cases the reader is advised to implement carefully these methods to avoid strange behavior (i.e., freeing object's attributes, modifying the current solution in a *local search* procedure, etc).

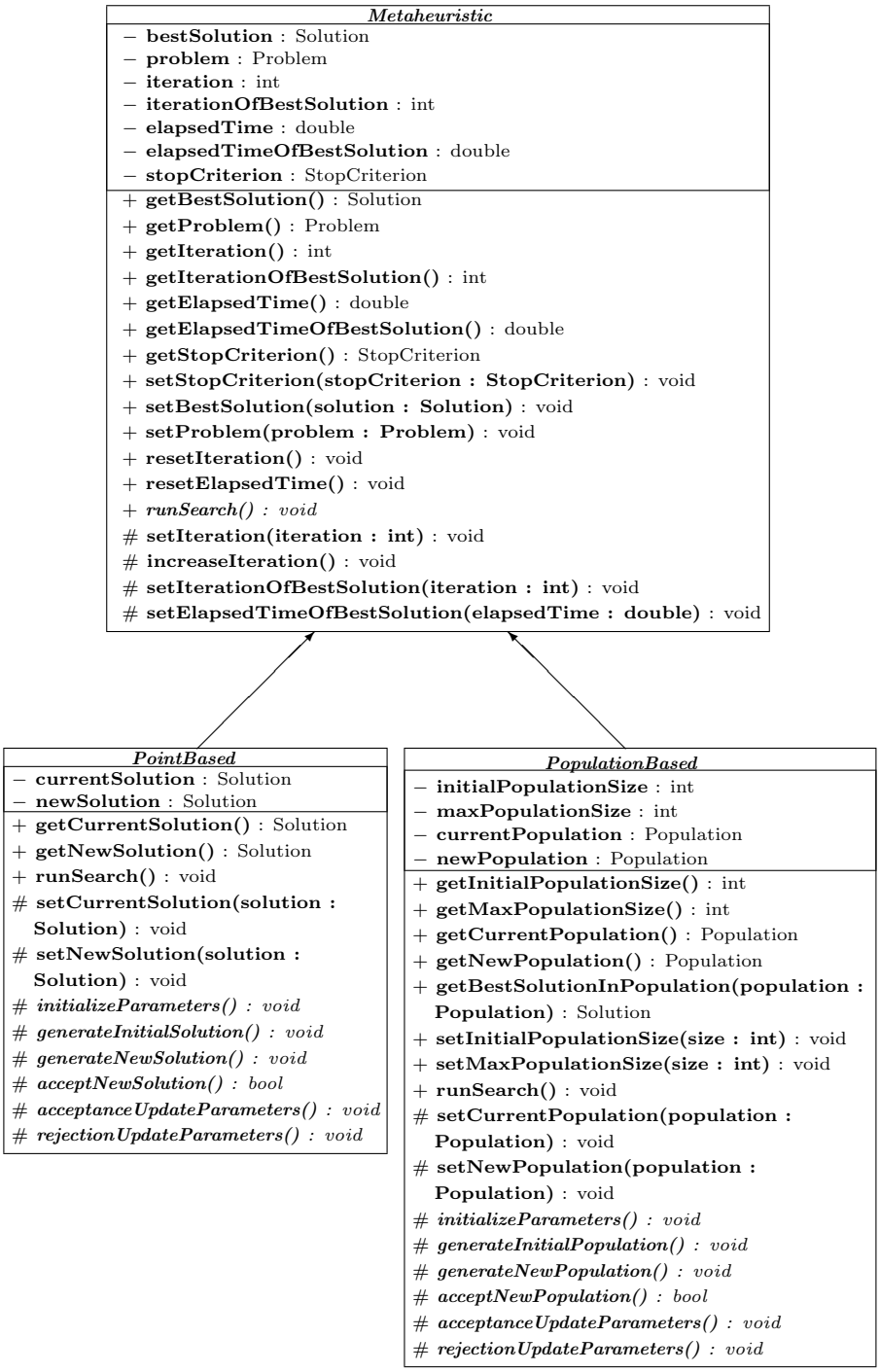

Figure 1.3. Class Hierarchy for the proposed metaheuristics

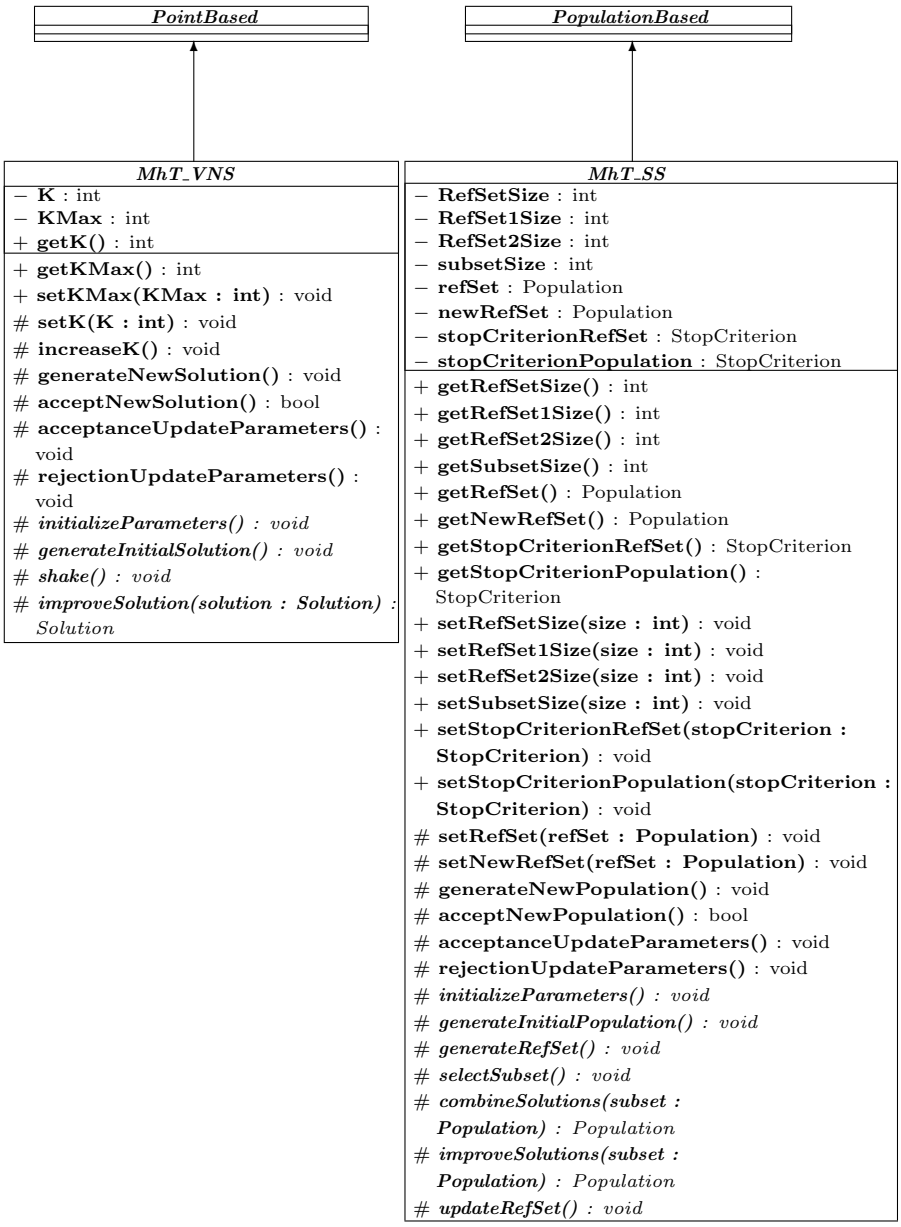

Figure 1.4. Class Hierarchy for the proposed metaheuristics (cont.)

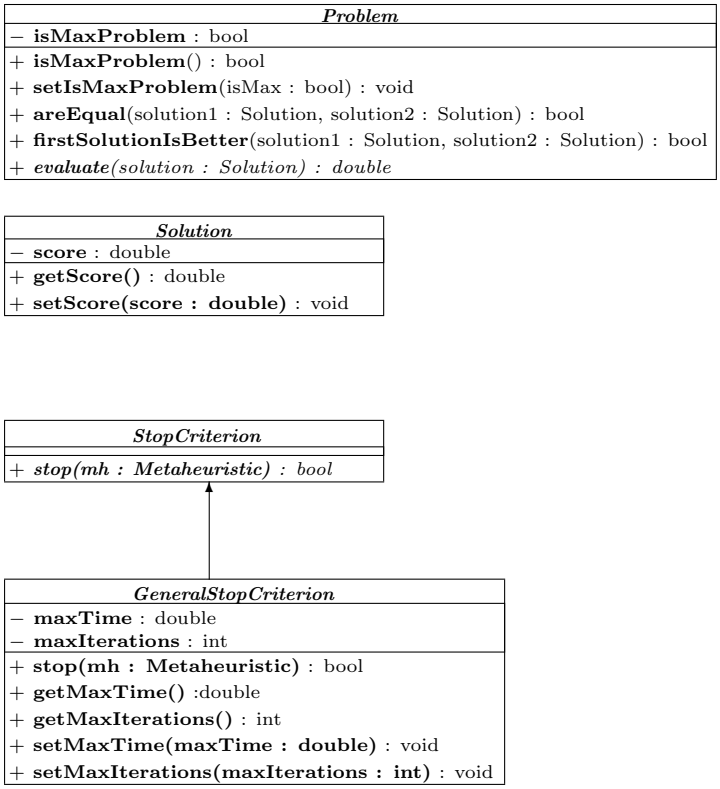

Figure 1.5. Class Hierarchy for the proposed metaheuristics (cont.)

2.1 Class Problem

Problem
− **isMaxProblem** : bool
+ **isMaxProblem**() : bool + **setIsMaxProblem**(isMax : bool) : void + **areEqual**(solution1 : Solution, solution2 : Solution) : bool + **firstSolutionIsBetter**(solution1 : Solution, solution2 : Solution) : bool + ***evaluate****(solution : Solution) : double*

The *Problem* class is probably the most important of all, but obviously, it is also the most problem-dependent one. Of course, there is no doubt that as class named *Problem* should be strongly problem-dependent. And that is why it is so difficult to generalize: we cannot assume anything about the attributes it contains, because they depend on the type of problem we are talking about. We can only assume that the problem should contain some kind of structure which explicitly enumerates all the elements that can form the solution, or at least, a way to obtain them. But that is all, we cannot know *a priori* if it is a list of objects, a function to obtain them, if we need more attributes to completely describe the problem... So, at most, we can define one attribute and some methods, which are listed below:

Attributes:

- **isMaxProblem**. Attribute that determines if the problem is a maximization or minimization problem.

Methods:

- **isMaxProblem**(). Method to get the value of the attribute *isMaxProblem*.
- **setIsMaxProblem**(isMax : bool). Method to set the value of the attribute *isMaxProblem*
- **areEqual**(solution1 : Solution, solution2 : Solution). Method to compare if two solutions have the same score. If a solution has not been evaluated yet, the method should call *evaluate* (explained below) and save the score obtained into the solution, then compare. The method should return *true* only if both solutions have the same score, and *false* in any other case.
- **firstSolutionIsBetter**(solution1 : Solution, solution2 : Solution). Similar to the method *areEqual* described above, but it checks if

the first solution has a higher score than the second one. The method should return *true* only if the first solution has an strictly higher score than the second one.

- ***evaluate****(solution : Solution)*. Abstract method to evaluate (give a score) to a solution. This method should be implemented by the subclass that specifies the problem.

2.2 Class Solution

Solution
− **score** : double
+ **getScore()** : double + **setScore(score : double)** : void

The *Solution* class is as problem-dependent as the *Problem* one, because it has to provide a correct arrangement for some elements of the problem in order to conform a solution to it. And that is exactly why it cannot be generally defined with much detail. We cannot know how this elements should be placed, it could be linearly, in which case we would use a vector or an array, or maybe it could need a more complex structure like a tree or a priority queue.

So, the specific structure to handle the elements must be left to a subclass that knows more about the problem. The only property that we think any solution should have, at least, is a representative value of the fitness of the solution.

Attributes:

- **score**. Attribute that reflects the fitness of the solution. This attribute is intended as a variable to store the value returned by the method *evaluate* of the class *Problem*. If the solution has not been yet evaluated, it should contain a *not a number* (NaN) value.

Methods:

- **getScore**(). Method to get the value of the attribute *score*.
- **setScore**(score : double). Method to set the value of the attribute *score*.

2.3 Class Metaheuristic

Metaheuristic

− **bestSolution** : Solution − **problem** : Problem − **iteration** : int − **iterationOfBestSolution** : int − **elapsedTime** : double − **elapsedTimeOfBestSolution** : double − **stopCriterion** : StopCriterion
+ **getBestSolution()** : Solution + **getProblem()** : Problem + **getIteration()** : int + **getIterationOfBestSolution()** : int + **getElapsedTime()** : double + **getElapsedTimeOfBestSolution()** : double + **getStopCriterion()** : StopCriterion + **setStopCriterion(stopCriterion : StopCriterion)** : void + **setBestSolution(solution : Solution)** : void + **setProblem(problem : Problem)** : void + **resetIteration()** : void + **resetElapsedTime()** : void + ***runSearch()*** *: void* # **setIteration(iteration : int)** : void # **increaseIteration()** : void # **setIterationOfBestSolution(iteration : int)** : void # **setElapsedTimeOfBestSolution(elapsedTime : double)** : void

This class is the base class from which every metaheuristic will inherit. It has a few attributes to control the execution of the metaheuristic, and a main method *runSearch* to look for a solution to a problem.

The sequence of use of this class in the general case should be as follows:

1 Inform the metaheuristic class of the problem we are considering by setting the *problem* attribute to the appropriate value.

2 Set a stop criterion for the search.

3 Call the *runSearch* method.

4 When the search is finished, get the best solution found.

This sequence can be altered in special cases, for example, when we already have a solution for the problem and we want the metaheuristic to improve it. In that case, before calling *runSearch*, we would have to set the *bestSolution* attribute to the solution object we have. The metaheuristic should then return a solution which is, at least, as good as the one provided, if not better.

The class also has attributes for posterior statistical analysis, such as the number of iterations run, the iteration in which the best solution was found, or the elapsed time of search until that moment.

Attributes:

- **bestSolution**. Attribute to store the best solution found until the moment. Normally, this attribute will be unset before the beginning of the search, but if a solution is provided, the metaheuristic should try to *continue* the search of the best solution from that point. In any case, the search method should not update the solution unless the new solution found has a higher score than the one provided.
- **problem**. This is an object containing the problem which we are searching for a solution.
- **iteration**. The current iteration of the *runSearch* main loop. The metaheuristic should reset this value to 0 each time the method *runSearch* is called.
- **iterationOfBestSolution**. Iteration in which the best solution was found.
- **elapsedTime** : A time-stamp for several usages. Normally, this attribute would be reset before the beginning of the search and will be updated at the finish of the search, containing the number of time units since the last reset ($search\_stop\_time - reset\_time$).
- **elapsedTimeOfBestSolution**. A time-stamp for the moment in which the best solution was found ($best\_solution\_time - reset\_time$).
- **stopCriterion**. Object to determine if the search should stop at a given moment or should continue searching for a better solution.

Methods:

- **getBestSolution**(). Method to get the best solution found by the metaheuristic.
- **getProblem**(). Method to get the problem object.
- **getIteration**(). Method to get the current iteration of the search.
- **getIterationOfBestSolution**(). Method to get the iteration of the best solution. If a best solution hasn't yet been found, it should return a non-numeric value.

- **getElapsedTime**(). Method to get the elapsed time (in time units) since the last reset.
- **getElapsedTimeOfBestSolution**(). Method to get the elapsed time (in time units) since the las reset until the moment the best solution was found.
- **getStopCriterion**(). Method to get the *StopCriterion* object of the metaheuristic.
- **setStopCriterion**(stopCriterion : StopCriterion). Method to set the *StopCriterion* object of the metaheuristic.
- **setBestSolution**(solution : Solution). Method to set the best solution found until the moment (for example, to continue a search).
- **setProblem**(problem : Problem). Method to set the *Problem* object.
- **resetIteration**(). Method to reset the iterations for the search.
- **resetElapsedTime**(). Method to reset the time from which we will count.
- ***runSearch()***. Abstract method to search for a solution. This method must be implemented by a subclass. The implementation should also reset the iterations and the elapsed time at the beginning of the method.
- **setIteration**(iteration : int). Method to set the current iteration. This method can only be called by an object of class Metaheuristic or subclass of it. An external object should never update this variable.
- **increaseIteration**(). Method to increase the current iterations.
- **setIterationOfBestSolution**(iteration : int). Method to set the iteration in which the best solution was found.
- **setElapsedTimeOfBestSolution**(elapsedTime : double). Method to set the time in which the best solution was found.

2.4 Class PointBased

PointBased

– **currentSolution** : Solution
– **newSolution** : Solution

+ **getCurrentSolution()** : Solution
+ **getNewSolution()** : Solution
+ **runSearch()** : void
# **setCurrentSolution(solution : Solution)** : void
# **setNewSolution(solution : Solution)** : void
# ***initializeParameters()*** *: void*
# ***generateInitialSolution()*** *: void*
# ***generateNewSolution()*** *: void*
# ***acceptNewSolution()*** *: bool*
# ***acceptanceUpdateParameters()*** *: void*
# ***rejectionUpdateParameters()*** *: void*

The *PointBased* class is one of the two subclasses of *Metaheuristic* we will implement. This class implements its methods taking in mind that a point-based metaheuristic will only obtain one solution per iteration, and if it is better than the best it has found, it updates this best one with the new solution obtained.

This schema represents the core of this class, and it is shown mainly in the *runSearch* method, that was abstract in the *Metaheuristic* class, and is now defined in this class to follow the former guidelines. The *runSearch* method calls several internal methods (see fig 1.6) that are declared abstract, in order to permit a subclass to define them in a way that matches the metaheuristic specific algorithm.

Attributes:

- **currentSolution**. Solution with which the metaheuristic is currently working.
- **newSolution**. Temporary variable in which the newly created solution is stored. If after generating a new solution it is accepted, then the current solution is replaced by the new one.

Methods:

- **getCurrentSolution**(). Method to get the current solution of the metaheuristic.
- **getNewSolution**(). Method to get the newly created solution in each iteration of the metaheuristic.
- **runSearch**(). Implementation of the method to search for a solution, specially adapted to point-based metaheuristics, in which some abstract methods are used.
- **setCurrentSolution**(solution : Solution). Method to set the current solution.

PointBased::runSearch

```
void PointBased::runSearch()
{
resetEllapsedTime();
resetIterations();
generateInitialSolution();
do {
    generateNewSolution();
    if (acceptNewSolution()) {
        acceptanceUpdateParameters();
        setCurrentSolution (getNewSolution());
        if (getProblem().firstSolutionIsBetter
            (getCurrentSolution(),getBestSolution()))
        {
            setTimeOfBestSolution(time());
            setIterationOfBestSolution(getIteration());
            setBestSolution(getCurrentSolution());
        }
    } else {
        rejectionUpdateParameters();
    }
    increaseIteration();
} while (!this.getStopCriterion().stop());
}
```

Figure 1.6. PointBased::runSearch code

- **setNewSolution**(solution : Solution). Method to set the newly created solution in each iteration of the metaheuristic.

- ***initializeParameters()***. Abstract method to initialize some parameters. The method is declared abstract, as we cannot know *a priori* how the metaheuristic needs to be initialized. A subclass that implements a specific metaheuristic, should implement also this method.

- ***generateInitialSolution()***. Abstract method to generate the initial solution. It is declared abstract, because the way in which an initial solution has to be generated depends not only on the metaheuristic, but also on the problem. Anyway, when this method is implemented, it should store the new solution in the *bestSolution* attribute. And also, if the *bestSolution* attribute is already set (for example, when we want the metaheuristic to continue a search from a solution), the method should not modify this value, but return immediately, leaving the attribute "as is".

- ***generateNewSolution()***. Abstract method to generate a new solution when a previous solution exists. The method is abstract for the same reason as the previous method *generateInitialSolution*. This method should store the new solution in the attribute *newSolution*, and if it needs the previous existing solution, it can access it through the *currentSolution* attribute.

- ***acceptNewSolution()***. Abstract method to decide if the newly created solution is accepted to become the current solution. This method is provided because some metaheuristics accept every new solution, but others check the new solution for some properties, and do not accept it if it does not conform to them.

- ***acceptanceUpdateParameters()***. Abstract method to call when the new solution is accepted.

- ***rejectionUpdateParameters()***. Abstract method to call when the new solution is rejected.

2.5 Class PopulationBased

PopulationBased

– **initialPopulationSize** : int
– **maxPopulationSize** : int
– **currentPopulation** : Population

− **newPopulation** : Population

+ **getInitialPopulationSize()** : int
+ **getMaxPopulationSize()** : int
+ **getCurrentPopulation()** : Population
+ **getNewPopulation()** : Population
+ **getBestSolutionInPopulation(population : Population)** : Solution
+ **setInitialPopulationSize(size : int)** : void
+ **setMaxPopulationSize(size : int)** : void
+ **runSearch()** : void
# **setCurrentPopulation(population : Population)** : void
# **setNewPopulation(population : Population)** : void
# ***initializeParameters()*** *: void*
# ***generateInitialPopulation()*** *: void*
# ***generateNewPopulation()*** *: void*
# ***acceptNewPopulation()*** *: bool*
# ***acceptanceUpdateParameters()*** *: void*
# ***rejectionUpdateParameters()*** *: void*

This class is the counterpart of the *PointBased* class, but for populations of solutions. These classes (*PointBased* and *PopulationBased*) were specifically designed to be as similar as possible, so that a parallelism between them could be naturally established. For example, if the class *PointBased* has a method called *generateNewSolution*, the class *PopulationBased* should have its equivalent called *generateNewPopulation*.

In Figure 1.7 we can see the code of the method *runSearch* for the *PopulationBased* class, and there is shown clearly the strong similarities that exist between this method and the respective one from *PointBased* class (1.6).

Just one more comment. In this class (and implicitly in all of its subclasses), there is an assumption for the object/type *Population*. For implementation purposes, we can simply consider it as an array, vector, list, etc. of *Solution* objects. The only requirements are that it can handle a set of solutions, granting the insertion, access and removal of each of them.

Attributes:

- **initialPopulationSize**. Number of elements (solutions) that should be contain in the initial population. The method to generate it, though abstract, should honor this value when implemented.
- **maxPopulationSize**. The maximum number of elements (solutions) that any population should contain. This value has to be observed every time a new population is created.

PopulationBased::runSearch

```
void PopulationBased::runSearch()
{
Solution bestInPopulation;

resetEllapsedTime();
resetIterations();
generateInitialPopulation();
do {
    generateNewPopulation();
    if (acceptNewPopulation()) {
        acceptanceUpdateParameters();
        setCurrentPopulation (getNewPopulation());
        bestInPopulation = getBestSolutionInPopulation
            (getCurrentPopulation());
        if (getProblem().firstSolutionIsBetter
            (bestInPopulation, getBestSolution()))
        {
            setTimeOfBestSolution(time());
            setIterationOfBestSolution(getIteration());
            setBestSolution(bestInPopulation);
        }
    } else {
        rejectionUpdateParameters();
    }
    increaseIteration();
} while (!this.getStopCriterion().stop());
}
```

Figure 1.7. PopulationBased::runSearch code

- **currentPopulation**. The population the metaheuristic is working on in the current iteration.
- **newPopulation**. The new population generated in each iteration of the metaheuristic. As with the *PointBased* class, if this method requires the previous population, it can access it through the *currentPopulation* attribute.

Methods:

- **getInitialPopulationSize**(). Method to get the size of the initial population.
- **getMaxPopulationSize**(). Method to get the maximum size of any population.
- **getCurrentPopulation**(). Method to get the population the metaheuristic is currently working on.
- **getNewPopulation**(). Method to get the new population created in each iteration of the metaheuristic.
- **getBestSolutionInPopulation**(population : Population). Method that looks for the solution with the highest score among all the population, and then returns it.
- **setInitialPopulationSize**(size : int). Method to set the attribute *initialPopulationSize*.
- **setMaxPopulationSize**(size : int). Method to set the attribute *maxPopulationSize*.
- **runSearch**(). Implementation of the method to search for a solution, specially adapted to population-based metaheuristics, in which in each iteration the metaheuristic explores a set of solutions. As in the *PointBased* class, some abstract methods are used.
- **setCurrentPopulation**(population : Population). Method to set the attribute *currentPopulation*.
- **setNewPopulation**(population : Population). Method to set the attribute *newPopulation*.
- ***initializeParameters()***. Abstract method to initialize some metaheuristic-dependent parameters.

- ***generateInitialPopulation()***. Abstract method to generate the initial population. As its *PointBased* counterpart, it is declared abstract, because the way in which an initial population has to be generated depends on the problem. If the *bestSolution* attribute is already set (for example, when we want the metaheuristic to continue a search from a solution), this solution should be included, or at least used to generate, the initial population.

- ***generateNewPopulation()***. Abstract method to generate a new population when a previous population exists. As the method *generateNewSolution* of the *PointBased* class, this method should store the new population in the attribute *newPopulation*, and if it needs the previous existing population, it can access it through the *currentPopulation* attribute.

- ***acceptNewPopulation() : bool***. Method to determine if the new population is accepted to substitute the current population. Usually, populations must have some properties in order to avoid degeneration of the solutions, and if it does not, the population is rejected.

- ***acceptanceUpdateParameters() : void***. Abstract method to call when the new population is accepted.

- ***rejectionUpdateParameters() : void***. Abstract method to call when the new population is rejected.

2.6 Class MhT_VNS

MhT_VNS
− **k** : int − **kMax** : int
+ **getK()** : int + **getKMax()** : int + **setKMax(kMax : int)** : void # **setK(k : int)** : void # **increaseK()** : void # **generateNewSolution()** : void # **acceptNewSolution()** : bool # **acceptanceUpdateParameters()** : void # **rejectionUpdateParameters()** : void # ***initializeParameters() : void*** # ***generateInitialSolution() : void*** # ***shake() : void*** # ***improveSolution(solution : Solution) : void***

This class is the first one we will see that represents an approximation to an specific metaheuristic. The *MhT_VNS* class uses attributes and methods that are exclusive of the *VNS* metaheuristic, although it is still declared *abstract* because of problem-dependent issues and the existence of several variances of the *VNS* general algorithm.

This class is intended to provide the general schema of the *VNS* metaheuristic, but at the same time, allow a subclass to customize some aspects of the algorithm, like, for example, the local search (here called *improveSolution*) or the shake procedure (see Figure 1.1). For example, a subclass of *MhT_VNS* could implement the *improveSolution* as an strictly local search, other could use a global search, and other could even use another metaheuristic.

For implementing this class the key concepts are, as in every other metaheuristic, to identify firstly if it is a *point based* or a *population based* metaheuristic, and secondly, where do the algorithm fit in the methods provided by the super-class (in this case, *PointBased*).

For example, the *generateNewSolution* method could consist in a *shake* and a *local search* (see Figure 1.8). The method to test if a new solution is accepted is simply a comparison between the new solution and the best solution found until the moment, and if has a higher score, it is accepted (Figure 1.9). If a solution is accepted, K is reset to 1 (Figure 1.10), and if it is rejected, K is increased (Figure 1.11).

MhT_VNS::generateNewSolution

```
void MhT_VNS::generateNewSolution()
{
    shake();
    improveSolution();
}
```

Figure 1.8. MhT_VNS::generateNewSolution code

Attributes:

- **k**. This attribute controls the current size of the neighborhood of the solution to explore in the *improveSolution* phase. This attribute varies from 1 to *kMax*.
- **kMax**. Maximum value for the k attribute.

Methods:

MhT_VNS::acceptNewSolution

```
void MhT_VNS::acceptNewSolution()
{
    return getProblem().firstSolutionIsBetter
        (getNewSolution(), getCurrentSolution());
}
```

Figure 1.9. MhT_VNS::acceptNewSolution code

MhT_VNS::acceptanceUpdateParameters

```
void MhT_VNS::acceptanceUpdateParameters()
{
    setK( 1 );
}
```

Figure 1.10. MhT_VNS::acceptanceUpdateParameters code

MhT_VNS::rejectionUpdateParameters

```
void MhT_VNS::rejectionUpdateParameters()
{
    increaseK();
}
```

Figure 1.11. MhT_VNS::rejectionUpdateParameters code

- **getK**(). Method to get the attribute *k*.
- **getKMax**(). Method to get the attribute *kMax*.
- **setKMax**(kMax : int). Method to set the attribute *kMax*.
- **setK**(k : int). Method to set the attribute *k*.
- **increaseK**(). Method to increase the value of the attribute *k*.
- **generateNewSolution**(). Method to generate the new solution from an existent one. See Figure 1.8.
- **acceptNewSolution**(). Method to determine if a new solution is accepted. See Figure 1.9.
- **acceptanceUpdateParameters**(). Method to call in case a new solution is accepted. See Figure 1.10.
- **rejectionUpdateParameters**(). Method to call in case a new solution is rejected. See Figure 1.11.
- ***initializeParameters()***. Abstract method to initialize parameters. This methods has to be implemented by a subclass that knows more details about the problem.
- ***generateInitialSolution()***. Abstract method to generate an initial solution. This methods has to be implemented by a subclass that knows more details about the solution.
- ***shake()***. Abstract method to provide the metaheuristic with a way to escape from a local minimum solution. It is declared abstract to allow a subclass to implement different ways of *shaking*, and also, because to be able to shake a solution, the method needs to know more details about the solution.
- ***improveSolution****(solution : Solution)*. Abstract method to improve a solution within a *k*-sized neighborhood. Like the *shake* method, this method is defined abstract both for allowing several implementations and because there is a need for more information on the problem and the solution. For a possible implementation of a local search procedure, see Figure 1.13.

2.7 Class MhT_SS

MhT_SS
– **refSetSize** : int – **refSetSize1** : int – **refSetSize2** : int – **subsetSize** : int – **refSet** : Population – **newRefSet** : Population – **stopCriterionRefSet** : StopCriterion – **stopCriterionPopulation** : StopCriterion
+ **getRefSetSize()** : int + **getRefSetSize1()** : int + **getRefSetSize2()** : int + **getSubsetSize()** : int + **getRefSet()** : Population + **getNewRefSet()** : Population + **getStopCriterionRefSet()** : StopCriterion + **getStopCriterionPopulation()** : StopCriterion + **setRefSetSize(size : int)** : void + **setRefSetSize1(size : int)** : void + **setRefSetSize2(size : int)** : void + **setSubsetSize(size : int)** : void + **setStopCriterionRefSet(stopCriterion : StopCriterion)** : void + **setStopCriterionPopulation(stopCriterion : StopCriterion)** : void # **setRefSet(refSet : Population)** : void # **setNewRefSet(refSet : Population)** : void # **generateNewPopulation()** : void # **acceptNewPopulation()** : bool # **acceptanceUpdateParameters()** : void # **rejectionUpdateParameters()** : void # ***initializeParameters()*** *: void* # ***generateInitialPopulation()*** *: void* # ***generateRefSet()*** *: void* # ***selectSubset()*** *: void* # ***combineSolutions(subset : Population)*** *: Population* # ***improveSolutions(subset : Population)*** *: Population* # ***updateRefSet()*** *: void*

This class is the other example of an specific metaheuristic we will see, but, as we mentioned earlier, instead of being *point based*, as was *VNS*, this metaheuristic is classified as *population based*. It is also defined abstract because, to be able to create an instantiable class, we need more information about the problem.

To implement this class, the first thing we have to do is find the methods of the *PopulationBased* class in which to insert the *Scatter Search*

specific code (see Figure 1.2 for the pseudocode). We only have to take in mind that the main loop for the *runSearch* method in *PopulationBased* consists in generating a new population in each iteration. Remember that the *Scatter Search* is based in the generation and update of a reference set in each iteration, not the population itself. This means that we have to think a little how to split the code of the algorithm in order to fit in the abstract methods used by *runSearch*.

An implementation of the classical *Scatter Search* will typically let the reference set evolve, and when it is done, return the best solution in it. So, in practice, it only uses one population. This fact has some implications, like, for example, that most of the code of the algorithm has to be embedded in the *generateNewPopulation* method. Another consequence is that other methods and attributes are meaningless, like *acceptNewPopulation* (which shall always return *true*), or the *Metaheuristic* attribute *stopCriterion* (which, as it refers to the evolution of the population, we want it to stop in the first iteration, so in fact, it has to return always *true*). More sophisticated implementations may use other stop criterions that would allow also the evolution of populations. For further reference on *Scatter Search* implementations, see [24].

MhT_SS::generateNewPopulation

```
void MhT_SS::generateNewPopulation()
{
    Population subset;
    do {
        setNewRefSet(new Population());
        do {
            subset = selectSubset();
            subset = combineSolutions(subset);
            subset = improveSolutions(subset);
            setNewRefSet(getNewRefSet().add(subset));
        } while(!getStopCriterionRefSet().stop());
        updateRefSet();
    } while(!getStopCriterionPopulation().stop());
}
```

Figure 1.12. MhT_SS::generateNewPopulation code

Attributes:

- **refSetSize**. Attribute that determines the size of the complete reference set (*good* solutions + *diverse* solutions).

- **refSetSize1**. Attribute that determines the number of *good* solutions that will be in the reference set.
- **refSetSize2**. Attribute that determines the number of *diverse* solutions that will be in the reference set.
- **subsetSize**. Attribute to determine the number of solutions that will be taken from the reference set to be combined. A typical value for this attribute is 2.
- **refSet**. Object containing the reference set of solutions.
- **newRefSet**. Object containing the new reference set of solutions generated in each iteration.
- **stopCriterionRefSet**. Object representing the stop criterion for the loop in which the new reference set is generated. When this stop criterion determines that the loop should stop, it will have generated a new reference set.
- **stopCriterionPopulation**. Object representing the stop criterion for the loop in which new reference sets are being generated. When this stop criterion determines that the loop should stop (usually because the new reference sets generated lack of good solutions or diverse solutions), a new reference set will have to be created from the population.

Methods:

- **getRefSetSize**(). Accessor method to get the value of the attribute *refSetSize*.
- **getRefSetSize1**(). Accessor method to get the value of the attribute *refSetSize1*.
- **getRefSetSize2**(). Accessor method to get the value of the attribute *refSetSize2*.
- **getSubsetSize**(). Accessor method to get the value of the attribute *subsetSize*.
- **getRefSet**(). Accessor method to get the value of the attribute *refSet*.
- **getNewRefSet**(). Accessor method to get the value of the attribute *newRefSet*.

- **getStopCriterionRefSet**(). Accessor method to get the value of the attribute *stopCriterionRefSet*.
- **getStopCriterionPopulation**(). Accessor method to get the value of the attribute *stopCriterionPopulation*.
- **setRefSetSize**(size : int). Accessor method to set the attribute *refSetSize*.
- **setRefSet1Size**(size : int). Accessor method to set the attribute *refSet1Size*.
- **setRefSet2Size**(size : int). Accessor method to set the attribute *refSet2Size*.
- **setSubsetSize**(size : int). Accessor method to set the attribute *subsetSize*.
- **setStopCriterionRefSet**(stopCriterion : StopCriterion). Accessor method to set the attribute *stopCriterionRefSet*.
- **setStopCriterionPopulation**(stopCriterion : StopCriterion). Accessor method to set the attribute *stopCriterionPopulation*.
- **setRefSet**(refSet : Population). Accessor method to set the attribute *refSet*.
- **setNewRefSet**(refSet : Population). Accessor method to set the attribute *newRefSet*.
- **generateNewPopulation**(). Method to generate a new population for the search. This method contains most of the code of the algorithm, because *Scatter Search* is based more in the evolution of the reference set than the evolution of the population. So, this method also uses other methods that will be explained bellow, like *combineSolutions*, or *updateRefSet*. See Figure 1.12 for the code of this method.
- **acceptNewPopulation**(). This method decides if a new population is accepted or not. In the classical implementation of *Scatter Search*, the process of finding a solution is based on the evolution of the reference set, not the population. So, in the practice, this method does not really has to check if the new population is better than the older, it should always accept the new one.
- **acceptanceUpdateParameters**(). Method to perform the needed operations when a population is accepted. As no new population is intended to be created, this method should contain no code.

- **rejectionUpdateParameters**(). This method is like the *acceptanceUpdateParameters* above, so it should contain no code.
- ***initializeParameters()***. Abstract method to initialize different variables that may be needed by a subclass, depending on the specific problem.
- ***generateInitialPopulation()***. Abstract method to generate the initial population that has to be defined by a subclass, as it may need problem-specific data.
- ***generateRefSet()***. Abstract method to generate a reference set from a population of solutions. This method should generate a set of size *refSetSize*, composed of *refSetSize1* good solutions and *refSetSize2* diverse solutions. A way to measure the diversity of a solution will also have to be implemented by a subclass.
- ***selectSubset()***. Abstract method to get the next subset of solutions from the reference set to be combined and improved. The subset selected should contain *subsetSize* solutions.
- ***combineSolutions****(subset : Population)*. Abstract method to combine the subset of solutions selected by the previous method. This method should produce a new solution from the subset.
- ***improveSolutions****(subset : Population)*. Abstract method to improve the new subset of solutions obtained by the method *combineSolutions*.
- ***updateRefSet()***. Abstract method that has to decide which of the solutions of the new reference set created should replace some of the solutions of the old reference set.

2.8 Class StopCriterion

StopCriterion
+ ***stop(mh : Metaheuristic)*** *: bool*

The *StopCriterion* class is one of the most remarkable features of this implementation. When talking about a stop criterion from the traditional point of view, we should expect a method of a subclass of *Metaheuristic* to handle this stop condition, having knowledge of the specific metaheuristic and problem used.

In the context of OOP, we have used a class approach instead. The explanation is simple: think for example of a test in which we wanted to compare the effectiveness of different metaheuristics, and we wanted to stop their search after 1 second. Why do we have to implement a subclass for every metaheuristic evaluated, repeating the same code for the stop criterion in each of them? This rises the risk of introducing errors, and at the same time, we are duplicating code, breaking the principles of encapsulation and modularity. All of this can be avoided by using a unique object *stop criterion* that will return *true* if the metaheuristic has been running for a second. This has several advantages:

- The *StopCriterion* class reduces drastically the number of subclasses that need to be created. If it were not for the *StopCriterion* class, we would have to create a subclass of each metaheuristic only to produce a different stopping condition of each of the loops in its search method. Think, for example, of the *MhT_SS* class, which uses at least, three different stop criterions. If we wanted to test just two stop criterions (for example, one based in the elapsed time, and another based in the number of iterations), that would imply, from the traditional approach, eight different subclasses of that metaheuristic to combine all the possibilities for the stop criterions. With the use of a *StopCriterion* class, there is no need for subclasses of the metaheuristic, this could be done simply by instantiating an object of each of the stop criterions and combining them in all the possible ways.
- The use of this class also increases the versatility of a metaheuristic, because with a sole implementation of this metaheuristic, its functionality can be fine-tuned in execution time, simply by changing its *stopCriterion* object. With the example of the *MhT_SS* class, is obvious that not only eliminates the necessity for subclasses of the metaheuristic, but also allows the *MhT_SS* class to exhibit different behaviors or "flavors" in execution time.

Of course, the use of the *StopCriterion* class is not exempt of disadvantages. For example, as the *StopCriterion* is an external class to *Metaheuristic* (i.e., it does not inherits from Metaheuristic), all the accessor methods to get the value of an attribute (the *getters*) need to be declared public in order to allow *StopCriterion* to check the values of the attributes. Moreover, we need to enable public accessors to variables that in other cases would normally not even exist, because the *StopCriterion* class may need access to some internal variables in order to determine if the stop condition is met. Anyway, despite this disadvantages, we recommend the use of this class for its benefits.

The *StopCriterion* class is defined abstract, with only one method, *stop()*, that has to be implemented by an specific subclass.

Methods:

- ***stop****(mh : Metaheuristic)*. Method to call when is needed to know if a metaheuristic should stop searching. This method receives a parameter, a *Metaheuristic* object, to which the *StopCriterion* will ask for some attributes in order to know if the stop condition is met.

2.9 Class GeneralStopCriterion

GeneralStopCriterion
− **maxTime** : double − **maxIterations** : int
+ **stop(mh : Metaheuristic)** : bool + **getMaxTime()** :double + **getMaxIterations()** : int + **setMaxTime(maxTime : double)** : void + **setMaxIterations(maxIterations : int)** : void

This class inherits from *StopCriterion*, and is aimed to determine a metaheuristic's stopping condition without depending on the problem nor the specific metaheuristic considered.

Before we continue, it is convenient to explain in more detail the differences among possible stopping criterions:

- **Problem-dependent stop criterions**. The stop criterion uses information that is specific of the problem, usually concerning the quality of the best solution obtained by the metaheuristic. For example, if we are dealing with a minimization problem, and we know which is the theoretical minimum value a solution can reach, we could stop searching when a solution is within a certain range above that minimum.
- **Metaheuristic-dependent stop criterions**. In this case, the stop criterion uses information about the metaheuristic itself. For example, in the *Scatter Search*, we need to stop some loops if the metaheuristic has reviewed all the specified combinations of solutions of the reference set, or when the quality of the solutions of the reference set lowers from a certain point.
- **Problem-Metaheuristic-dependent stop criterions**. This case is a mixture of the two previous cases, when the stop criterion

uses both problem-dependent and metaheuristic-dependent information. An example of this could be to stop the search of a *VNS* metaheuristic either when the quality of the solution is above a certain level, or when the value of K has reached the top ($KMax$) and has not been reseted to 1 for a long time (which could indicate that the solution is good, as we are always moving the maximum distance between neighborhoods without finding a better solution).

- **Independent stop criterions**. This is a generic case in which the stop criterion accesses information that do not depend on the problem nor the metaheuristic used. The typical information used here is the number of iterations of the metaheuristic or the elapsed time since the beginning of the search.

This class is an implementation of an *independent stop criterion*, that can be configured to use the iterations, the elapsed time, or both.

Attributes:

- **maxTime**. Attribute that determines the maximum time a metaheuristic is allowed to run. If this attribute is not going to be used, it should contain a NaN value.
- **maxIterations**. Similar to the attribute above, but it determines instead the maximum number of iterations a metaheuristic is allowed to run.

Methods:

- **stop**(mh : Metaheuristic). Method that returns *true* if the metaheuristic has been running for more than *maxTime* or has looped through more than *maxIterations* iterations.
- **getMaxTime**(). Method to get the attribute *maxTime*.
- **getMaxIterations**(). Method to get the attribute *maxIterations*.
- **setMaxTime**(maxTime : double). Method to set the attribute *maxTime*.
- **setMaxIterations**(maxIterations : int). Method to set the attribute *maxIterations*.

3. Implementation: The *p*-Median Problem

The p-selection problems constitute a wide class of hard combinatorial optimization problems whose solutions consist of p items from a universe

U. The standard moves for this class of problems are the interchange moves. An interchange move consists of replacing an item in the solution by another one out of the solution. A very representative p-selection problem is the p-median location problem whose standard version is explained below. The p-median problem is a well known location problem (see [26] or [4]) that have been proved $\mathcal{NP}$-hard [23]. This problem has often been used to test metaheuristics, among them parallel VNS [5] and Scatter Search [6].

Let $U = \{u_1, u_2, ..., u_n\}$ be the set of the locations of a finite set of users that are also potential locations for p facilities. Let D be the $n \times n$ matrix whose entries contain the distances $d_{ij} = D(u_i, u_j)$ between the points u_i and u_j, for $i, j = 1, ..., n$. The distance between a set of points $X \subset U$ and a point $u_i \in U$ is stated as follows:

$$D(X, u_i) = \min_{u_j \in X} D(u_j, u_i).$$

The cost function for a set of points X is the sum of the distances to all the points in U; i.e.,

$$f(X) = \sum_{u_i \in U} \min_{u_j \in X} D(u_j, u_i) = \sum_{u_i \in U} D(X, u_i).$$

The p medians selected from U constitute the set that minimizes this cost function. The optimization problem is then stated as follows:

$$\text{minimize} \sum_{u_i \in U} \min_{u_j \in X} D(u_i, u_j)$$

where $X \subseteq U$ and $|X| = p$.

Using a solution coding that provides an efficient way of implementing the moves and evaluating the solutions is essential for the success of any search method. A solution X can be represented by an array $x = [u_i : i = 1, ..., n]$ where u_i is the i-th element of the solution for $i = 1, 2, ..., p$, and the $(i - p)$-th element outside the solution for $i = p + 1, ..., n$. Let X_{ij} denote the solution obtained from X by interchanging u_i and u_j, for $i = 1, ..., p$ and $j = p + 1, ..., n$.

For the p-selection problems, as the p-median problem, the local search procedure is based on the explained interchange moves. The usual greedy local search is implemented by choosing iteratively the best possible move among all interchange moves. The code of a local search that can be used in the *improveSolution* method of the *MhT_VNS* class is given in Figure 1.13. Here, the function *improved* tests if the new solution improves the current one or not. The *exchange* method is defined in Figure 1.14

Local Search

```
void local_search(sol cur_sol)
{
 init_sol = cur_sol ;
 while improved(cur_sol,init_sol))) {
 for (i=p;i<n;i++)
   for (j=0;j<p;j++) {
      exchange(new_sol,cur_sol,i,j) ;
      if improved(new_sol,cur_sol)
        cur_sol = new_sol
      } /* for */
 } /* while */
} /* local_search */
```

Figure 1.13. Local Search Pseudocode

Exchange

```
void exchange(Solution new_sol, Solution cur_sol,
              int i, int j)
{
 facilities = cur_sol.getFacilities();
 aux = facilities[i] = facilities[j];
 facilities[i] = facilities[j];
 facilities[j] = aux;
 new_sol.setFacilities(facilities);
}
```

Figure 1.14. Exchange Pseudocode

In order to use the class hierarchy explained above to solve the p-median problem, we define the problem and solution objects for this problem. These classes inherit from their respective superclasses *Problem* and *Solution*, defined previously.

We call the problem class *PMedian_Problem* that is declared as follows:

PMedian_Problem

− **locations** : List
− **n** : int
− **p** : int

+ **getLocations()** : List
+ **getN()** : int
+ **getP()** : int
+ **setLocations(List : locations)** : void
+ **setN(n : int)** : void
+ **setP(p : int)** : void
+ **evaluate**(solution : Solution) : double

---

where *locations* is a list (vector, array, etc) of points, n is the size of that list, i.e., the number of possible locations, and p is the number of facilities we want to allocate. The methods are simply accessors to the attributes, except for the *evaluate* method, which calculates the sum of the distances of every point in the *locations* list to its nearest facility.

The solution class will be called *PMedian_Solution*, and will be defined as follows:

PMedian_Solution

− **facilities** : List

+ **getFacilities()** : List
+ **setFacilities(List : facilities)** : void

---

where the list *facilities* denotes the points where the facilities will be allocated. In this case, where the facility points are only allocated in the given location points, it is easier for all the functions to deal with the *PMedian_Solution* class if the *facilities* attribute contains all the possible locations, ordered in the way we mentioned above. That is, the first p elements of the list are the points selected for the facilities, and the final $n - p$ points are the discarded. The methods are simply accessors to the attributes.

3.1 VNS for the *p*-median

PMedian_VNS

initializeParameters() : void
generateInitialSolution() : void
shake() : void
improveSolution() : void

The basic VNS pseudocode 1.1 can be applied to any problem by providing the implementation of the initialization procedure, the shaking method, the local search and the function to test whether the solution is improved or not.

The *shake* procedure consists of, given the size k for the shake, choosing k times two points, u_i in the solution and u_j outside the solution at random, and performing the corresponding interchange move (see Figure 1.15).

The *improveSolution* method is implemented using the *Local Search* defined previously using the basic *Exchange* movement.

The *initializeParameters* method is void, although some initialization can be done here (for example, initializing a seed for a random number generation routine).

The *generateInitialSolution* method simply selects random points to be included in a solution, unless an initial solution is provided, in which case the initialization does nothing.

PMedian_VNS::shake

```
void PMedian_VNS::shake(sol cur_sol)
{
 init_sol = cur_sol ;
 for (r=0;r<k;r++) {
   i = rnd % p ;
   j = p + rnd % (n-p) ;
   exchange(cur_sol,new_sol,i,j) ;
   cur_sol = new_sol ;
 } /* for */
} /* shake */
```

Figure 1.15. Shake Pseudocode

3.2 Scatter Search for the *p*-median

PMedian_SS

− **subsetI** : int − **subsetJ** : int
+ **getSubsetI()** : int + **getSubsetJ()** : int # **setSubsetI**(i : int) : void # **setSubsetJ**(j : int) : void # **initializeParameters()** : void # **generateInitialPopulation()** : void # **generateRefSet()** : void # **selectSubset()** : void # **combineSolutions(subset : Population)** : Population # **improveSolutions(subset : Population)** : Population # **updateRefSet()** : void

The Scatter Search for the p-median problem uses the standard parameter setting and rules explained above. The key idea to apply the scatter search principles to an optimization problem is the distance between solutions used to evaluate the dispersion among them. This distance for the p-median problem is defined using the same objective function. Let $f_Y(X)$ be the objective function for the set of users in Y:

$$f_Y(X) = \sum_{v \in Y} \min_{u \in X} Dist(u, v)$$

The distance between two solutions X and Y is given by $Dist(X, Y) = f_Y(X) + f_X(Y)$.

We now summarize an implementation of the components of the Scatter Search for the p-median problem, explaining the key methods mentioned above.

1 **generateInitialPopulation.** A simple way to generate a population consists in randomly creating solutions. We select p times a new point from U that is successively included in the set X. Given the previously fixed size $PopSize$ of random solutions, the population Pop is obtained by applying the local search to each random solution as the improvement method. See Figure 1.16.

2 **generateRefSet.** To generate a reference set from the population we first include in $RefSet$ the $RefSetSize_1$ best solutions. Then we iteratively include in the $RefSet$ the farthest solution from the solutions already in $RefSet$, repeating this procedure $RefSetSize_2$ times. We then obtain the reference set $RefSet$ with size $RefSetSize = RefSetSize_1 + RefSetSize_2$. The code of this method is given in Figure 1.17. The *getFurthermostSolution*

method used in the code should return the furthermost solution of a given population from those in the reference set, using the *distance* concept defined above.

3 **selectSubset.** The selection of a subset to apply the combination consists in considering all the subsets of fixed size r (usually $r = 2$). Figure 1.18 contains the code of this method. This method maintains its main indexes as class attributes for allowing a *StopCriterion* class to determine if the subset generation loop has finished.

4 **combineSolutions.** The combination of each two solutions consists in the following. In the first place this method selects the points common to both solutions. Let X be the set of these points. For every point $u \in U \setminus X$ let

$$L(u) = \{v \in U : Dist(u, v) \leq \beta Dist_{max}\}$$

where

$$Dist_{max} = \max_{u,v \in U} Dist(u, v).$$

Choose the point $u^* \in U$ such that $Dist(X, u^*) = \max_{u \in U} Dist(X, u)$ and select at random a point $v \in L(u^*)$ that is included in X. This step is iteratively applied until $|X| = p$. The code of this method is shown in Figure 1.19. There, the method calls several procedures like *getCommonPoints*, that should return a solution with the points that are part of all the solutions; *getDiffPoints*, that does exactly the opposite, returning the points that do not appear in all the solutions; *getFurthermostPoint*, that returns the point with the largest distance to a set of points, selected from a set of possibles; finally, *selectNearNeighbour* returns a random point from a set of possibles that are considered "near" a given one (here, the limit of "near" is controled by the parameter β).

5 **improveSolutions.** Given a solution, the **improveSolutions** performs the local search on a population of solutions using the interchange moves.

6 **updateRefSet.** Let $ImpSolSet$ be the set of all the solutions reached by the **improveSolutions**. Apply **generateRefSet** to the set $RefSet \cup ImpSolSet$.

4. Conclusions

The ability of OOP to develop encapsulated, extensible software makes this paradigm one of the most suitable for programming metaheuristics.

PMedian_SS::generateInitialPopulation

```
void PMedian_SS::generateInitialPopulation()
{
 for (i=0;i<getInitialPopulationSize();i++) {
   cur_sol = generateRndSolution();
   cur_sol = improveSolution(curr_sol);
   getCurrentPopulation().add(cur_sol);
 }
}
```

Figure 1.16. generateInitialPopulation code

PMedian_SS::generateRefSet

```
void PMedian_SS::generateRefSet()
{
 // evaluate the solutions
 for (i=0;i<getCurrentPopulation().size();i++) {
   cur_sol = (curr_sol);
   getProblem().evaluate(getCurrentPopulation().get(i));
 }

 // order the solutions by their score
 sort(getCurrentPopulation());

 // add refSetSize1 best solutions to the refSet
 for (i=0;i<getRefSetSize1();i++) {
   getRefSet().add(getCurrentPopulation().get(i));
 }

 // add refSetSize2 distant solutions to the refSet
 for (i=0;i<getRefSetSize2();i++) {
   cur_sol = getFurthermostSol(population,refSet);
   refSet.add(cur_sol);
 }
}
```

Figure 1.17. generateRefSet code

PMedian_SS::selectSubset

```
Population PMedian_SS::selectSubset()
{
 i = getSubsetI();
 j = getSubsetJ();
 if (NaN(i) || NaN(j)) {
   i = 0;
   j = 1;
 } else if (j == getRefSetSize()-1) {
   i++;
   j = i + 1;
 }
 setSubsetI(i);
 setSubsetJ(j);
 return list(getCurrentPopulation.get(i),
     (getCurrentPopulation.get(j));
}
```

Figure 1.18. selectSubset code

PMedian_SS::combineSolutions

```
Population PMedian_SS::combineSolutions(Population solutions)
{
 newSolution = getCommonPoints(solutions);
 possibleSolutions = getDiffPoints(solutions);
 while (newSolution.getFacilities().size() <
        problem.getP()) {
   point = getFurthermostPoint(newSolution,
       possibleSolutions);
   point = selectNearNeighbour(possibleSolutions,
       point,beta);
   newSolution.getFacilities().add(point);
 }
 return list(newSolution);
}
```

Figure 1.19. combineSolutions code

Most metaheuristics appearing in the literature share some common aspects in their design (a main loop, attributes like the number of iterations, etc), which makes them good candidates for establishing a class hierarchy.

The hierarchy we have proposed tries to differentiate between *point-based* and *population-based* metaheuristics, but maintaining at the same time an intuitive parallelism between their respective methods, which makes them easier to understand. We have also proposed an implementation of a metaheuristic of each class, *VNS* for *point-based*, and *Scatter Search* for *population-based*. Although it is a very reduced set of examples, having in mind the number of metaheuristics currently created, we hope that they give a significant insight of how other metaheuristics could be implemented with this approach. The purpose is to make the reader able to program new metaheuristics without getting stuck in programming details, just caring of coding the relevant parts of the algorithm.

Another property of OOP is the encapsulation and mobility it provides to objects. For example, as we mentioned in the *MhT_VNS* class, we can produce an object to perform a local search in the *improveSolution* method, but we could perfectly use a metaheuristic object to do that task. This example illustrates how *objects* may help improving versatility and usability in metaheuristic's software design.

The reader may have also noted that, although the code provided has a $C++$-like style, we have tried not to stick to a specific programming language, since our purpose was the design of the class hierarchy, which is code-independent. The classes and their respective attributes and methods where designed to use commonly extended features and data types, in order to allow portability between languages.

As an additional comment the hierarchy proposed in this work has been successfully implemented in practice in the context of the *Weka Project* ([34], [36]). *Weka* is a collection of machine learning algorithms oriented to data mining tasks, that is implemented in Java, and that provides a graphical interface for dealing with the algorithms it contains. The project defines the interface any *Data Mining* class should have to be able to interoperate with the Weka environment, and all the classes included in the project conforming to the interface are accessible to the final user. This environment allows a user to graphically interact with these algorithms, including metaheuristics. This illustrates the power of encapsulated, top-down design, allowing a problem-oriented group of classes (like *Metaheuristic*) to be transparently integrated in a graphical environment.

Acknowledgments

This research has been partially supported by the Spanish Ministry of Science and Technology through the project TIC2002-04242-C03-01; 70% of which are FEDER funds.

The research of the co-author M. García Torres has been partially supported by a CajaCanarias grant.

References

[1] J. Belisle. OMG Standards for Object-Oriented Programming. AIXpert, pp. 38-42, Aug. 1993.

[2] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison *ACM Computing Surveys*, 35-3:268-308, 2003.

[3] M. Dorigo and T. Stutzle. The ant colony optimization metaheuristic: Algorithms applications, and advances. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*. 2003.

[4] Z. Drezner. *Facility location: A survey of applications and methods*, Springer, 1995.

[5] F. García-López, B. Melián-Batista, J.A. Moreno-Pérez, J.M. Moreno-Vega. The Parallel Variable Neighborhood Search for the p-Median Problem *Journal of Heuristics* 8 (2002) 375–388.

[6] F. García-López, B. Melián-Batista, J.A. Moreno-Pérez, J.M. Moreno-Vega. Parallelization of the scatter search for the p-median problem. *Parallel Computing* 29 (2003) 575–589.

[7] F. Glover. Heuristics for Integer Programming using Surrogate Constraints, *Decision Sciences* 8, (1977) 156–166.

[8] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[9] F. Glover. Tabu Search for Nonlinear and Parametric Optimization (with Links to Genetic Algorithms). *Discrete Applied Mathematics* 49 (1994) 231–155.

[10] F. Glover. A template for scatter search and path relinking. In J.-K. Hao and E. Lutton, editors, *Artificial Evolution*, volume 1363, pages 13–54. Springer-Verlag, 1998.

[11] F. Glover Scatter Search and Path Relinking. in D. Corne, M. Dorigo, F. Glover (Eds.) *New Ideas in Optimisation*, Wiley, (1999).

[12] F. Glover and G. Kochenberger (eds.). *Handbook of Metaheuristics*. Kluwer, 2003.

[13] F. Glover and M. Laguna. *Tabu Search.* Kluwer, 1997.

[14] F. Glover, M. Laguna, R. Martí. Fundamentals of Scatter Search and Path Relinking *Control and Cybernetics*, 39, (2000) 653-684.

[15] F. Glover, M. Laguna, R. Martí. Scatter Search. in *Theory and Applications of Evolutionary Computation: Recent Trends*, A. Ghosh, S. Tsutsui (Eds.) Springer-Verlag, (2003).

[16] P. Hansen, N. Mladenović. Variable Neighborhood Search for the p-Median. *Location Science* 5 (1997) 207–226.

[17] P. Hansen and N. Mladenović. An Introduction to Variable Neighborhood Search. in: S. Voss et al. eds., Metaheuristics, Advances and Trends in Local Search Paradigms for Optimization (Kluwer, 1999) 433-458.

[18] P. Hansen and N. Mladenović, Variable Neighborhood Search: Principles and applications, *European Journal of Operational Research* 130:449-467, 2001.

[19] P. Hansen and N. Mladenović. Developments in variable neighbourhood search. In C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 415–439. 2002.

[20] P. Hansen and N. Mladenović. Variable neighborhood search. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 221–234. Oxford University Press, 2002.

[21] P. Hansen, N. Mladenović. Variable Neighborhood Search. In F. Glover and G. Kochenberger (eds.), *Handbook of Metaheuristics* Chapter 6, 2003.

[22] D. Henderson, S.H. Jacobson and A.W. Johnson. Theory and Practice of Simulated Annealing. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*, chapter 10. 2003.

[23] O. Kariv, S.L. Hakimi. An algorithmic approach to network location problems; part 2. The p-medians. *SIAM Journal on Applied Mathematics*, 37(1969), 539-560.

[24] M. Laguna and R. Martí. *Scatter Search: Metodology and Implementations in C.* Kluwer Academic Press, (2003).

[25] J.A. Lozano and P. Larrañaga. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation.* Kluwer Academic, 2002.

[26] P. Mirchandani and R. Francis, (eds.). *Discrete location theory.* Wiley-Interscience, 1990.

[27] J.-Y. Potvin, K. Smith. Artificial Neural Networks for Combinatorial Optimization. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*, chapter 15. 2003.

[28] C.R. Reeeves. Genetic algorithms. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*, chapter 3. 2003.

[29] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Press, 1993.

[30] M. Resende, C. Ribeiro. Greedy Randomized Adaptive Search Procedures. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*, chapter 8. 2003.

[31] C.C. Ribeiro and P. Hansen, editors. *Essays and Surveys in Metaheuristics*, volume 15. Kluwer, 2002.

[32] E.A. Silver, R. Victor, V. Vidal, and D. De Werra. A tutorial on heuristic methods. *European Journal of Operational Research*, 5:153–162, 1980.

[33] C. Voudouris and E.P.K. Tsang. Guided local search. In F. Glover and G. Kochenberger, editors, *Handbook on MetaHeuristics*, chapter 7. 2003.

[34] Weka project webpage.
http://www.cs.waikato.ac.nz/ ml/weka/

[35] Wikipedia on-line definition for OOP.
http://en.wikipedia.org/wiki/Object-oriented_programming

[36] I.H. Witten, E. Frank Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations Morgan Kaufmann, 1999

[37] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. *Systems and Computers in Japan*, 32(3):33–55, 2001.

[38] M. Yagiura and T. Ibaraki. Local search. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 104–123. Oxford University Press, 2002.

[39] S.H. Zanakis, J.R. Evans, and A.A. Vazacopoulos. Heuristic methods and applications: a categorized survey. *European Journal of Operational Research*, 43:88–110, 1989.