A Framework for Memetic Algorithms

by

Fengjie Wu

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
in
Computer Science

University of Auckland

Supervisor: Dr. Michael J. Dinneen

ABSTRACT

Many optimization problems are fundamentally *hard*. Essentially, a 'hard' problem is one for which we cannot guarantee to find the best solution in a reasonable amount of time. In practice, however, the quest to solve hard problems is not quite so hopeless as this definition suggests. This is due to the use of *approximate methods*.

An approximate method is an algorithm that we use to try to find solutions to hard optimization problems, and which runs quickly, but which gives no guarantee that the solution it will find is the best one. The existing, successful methods in approximate optimization fall into two broad classes: *local search*, and *population-based search*. There are many *population-based* optimization algorithms and various ways to handle the optimization issues. In this thesis, a special emphasis has been given to *Memetic Algorithms* introduced by P. Moscato, which represents one of the most successful emerging ideas in the ongoing research effort to understand population based and local search algorithms.

Based on a general template for *Memetic Algorithms*, an object-oriented $framework$ was developed in $C{+}{+}$ to experiment with this approximate method. When the user wants to use this framework for a special NP-Hard optimization problem, he needs only to define suitable derived classes, which implement the virtual functions of the abstract classes, and supply the problem specific details.

Using and instantiating this *Memetic Algorithms framework*, two applications were encoded for two well-known combinatorial optimization problems, *Vertex Cover* and *Independent Set*. Comparisons of effectiveness were made between several approximate methods.

ACKNOWLEDGEMENTS

To Dr. Michael J. Dinneen, my supervisor, for the encouragement and guidance that he provided throughout the preparation of this thesis. His enthusiasm, knowledge insight and techniques helped me immeasurably.

To Dr. Pablo Moscato, the examiner of my thesis , for his time, working and invaluable suggestion.

To Sophie Duo, my love, the true engine of my thesis.

To my parents, for the support that they always supply.

To Juhua Zhou, for his friendship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Optimization

Optimization is a key topic in computer science, artificial intelligence, operational research, and related fields. Outside these scientific communities the real meaning of optimization is rather imprecise: it simply means "making better". However, in the context of this thesis, optimization is the process of trying to find the best possible solution to an *optimization problem* within a given time limit. In turn, an optimization problem is simply a problem for which there are different possible solutions, and there is some clear notion of solution quality. That is, an optimization problem exists when different candidate solutions can be meaningfully compared and contrasted.

There are a lot of such problems in industry and science. Some problems are very simple to address because there are very few possible candidate solutions, and we can simply assess each of them in turn. Or, there may be too many candidates for an exhaustive search, but a fundamental understanding of the problem will either yield the best solution directly, or a quick way to find it.

However, many optimization problems are fundamentally *hard*. This is the most typical scenario when it comes to realistic and relevant problems in industry or science. The intuitive idea of hardness corresponds to some quite precise ideas in computer science. Essentially, a 'hard' problem is one for which we cannot guarantee to find the best solution in an reasonable amount of time. In practice, however, the quest to solve hard problems is not quite so hopeless as this definition suggests. This is due to the use of *approximate methods*. An approximate method is an algorithm that we use to try to find solutions to hard optimization problems, and which runs quickly, but which gives no guarantee that the solution it will find is the best one. Good approximate methods, especially when tailored and customized to work with a particular optimization problem, may often find a good solution very quickly.

Since most real world optimization problems seem to be both fundamentally hard and also practically hard, research into good approximate methods remains valuable, and continues worldwide. Any new development in optimization that leads to better results on a particular problem, or to new approximate methods which may be applied to a wide range of problems, is of considerable value to science and industry. It may lead to substantial financial savings for corporations, significantly more effective health care, or appreciably cheaper and more reliable communications. This collection of ongoing motivations has led to a fairly steady stream of ideas for approximate algorithms to solve hard optimization problems. In particular, some of the new ideas over the past decade have given improved approaches to many well-known hard optimization problems [CDG99].

1.2 A NP-Optimization Problem and Its Approximate Solution

One of the most accepted ways to prove that a problem is hard is to prove it NP-complete. If a decision problem is NP-complete we are almost certain that it cannot be solved optimally in polynomial time.

If we are given an algorithm to solve a problem, we can often use it to solve other similar problems. Given two problems A and B, we can specify in advance how to use any algorithm for problem B to solve problem A. Such a specification is called a reduction from problem A to B. If we are able to prove that the process is correct, it is said that we have reduced A to B.

In the 1970s and 1980s, a lot of decision problems for which the only accepted answers are either "Yes" or "No" were proved to be reducible to each other. All these problems have a common property: for every input to a problem with a "Yes" solution there is a proof that the input has solution "Yes" and this proof can be verified in polynomial time. Any problem with this property is called an NP problem. We say this for decision problems, but that is easy to infer that all problems which can be solved by polynomial time algorithms satisfy this property. We say that $P \subseteq NP$ where P is the set of problems which can be solved by polynomial time algorithms and NP is the set of NP problems.

A problem which can be said to be "harder" than all NP problems, that is a problem to which every problem in NP be reduced, is called NP-hard. If an NP-hard problem is itself an NP problem it is called NP-complete. Thus all NP-complete problems are equally hard to solve, since they are inter-reducible. If there is a polynomial time algorithm for an NP-complete problem then $P = NP$ and every NP problem can be solved in polynomial time. Despite enormous efforts the question whether $P = NP$ is still unanswered. The common belief nowadays

is that $P \neq NP$ and a big part of the research in Theoretical Computer Science has $P \neq NP$ as a fundamental assumption.

So according to the above discussion, we realize that there are some problems that can not be approximated with efficient algorithms at all. On the other hand, it depends on the problem since we know that for some problems that are equally hard to solve optimally, some can be approximated very well with efficient algorithms. This makes it possible that some problems can be efficiently approximated using algorithms based on evolutionary techniques while others can not [Mos01].

1.3 Approximate Iterative Method

There are a handful of key types of optimization methods, ranging from specific to general. What we concentrate on in this thesis are approximate iterative methods. These methods usually take on the very general shape of the algorithm in *Figure* 1.1:

1. **Begin:** Generate and evaluate an initial collection of candidate solutions S.
2. **Operate:** Produce and evaluate a collection of new candidate solutions S' by making randomized changes to selected members of S.
3. **Renew:** Replace some of the members of S with some of the members of S', and then return to 2 unless some termination criterion has been reached.

Figure 1.1: Generalized iterative method.

Approximate iterative search methods are all based on the key idea of generate and test. Somehow, a candidate solution to the problem at hand is generated. It is then tested for quality. If it is found to be not good enough, then we repeat the process, each time generating a different candidate solution to test. The most obvious version of an iterative generation and test strategy is random search. At each iterate, we simply come up with a brand new random solution. However, a huge variety of optimization methods have been explored which basically follow the generate and test theme, but use more interesting and fruitful ways to guide the generation of new candidate solutions. In these methods, the key is somehow to use information from previously visited solutions to guide the generation of new candidates. Since the number of potential candidate solutions is very massively larger than the number we can hope to generate and test in reasonable time, the candidate solution generator component of an optimization method is critical to success.

To design an optimization method is, therefore, to design a way of dealing with the following question: How can we use information gained from previously seen

candidate solutions to guide the generation of new ones? In random search, we do not use information from previous solutions at all; performance of random search is consequently very poor. But in current, well-established optimization methods, three successful ideas in response to this question are mainly used. These are:

- **Idea 1:** New candidate solutions are slight variations of previously generated candidates.
- **Idea 2:** New candidate solutions are generated by recombining aspects of two or more existing candidates.
- **Idea 3:** The 'parent' from which new candidates are produced are selected from previously generated candidates via a stochastic and competitive strategy which generates better performing candidates.

The 'slight variations' mentioned in Idea 1 are achieved by so-called *neighborhood* or *mutation* operators, also called *move* operators. For example, suppose we are trying to solve a time-tabling problem and we encode a solution as a string of numbers such as that jth number represents the time slot given to event j. We can simply use a mutation operator which changes a randomly chosen number in the list to a new, randomly chosen value. The mutant timetable is thus a slight variation on its 'parent' timetable.

Operations which produce new candidates from two or more existing ones are called *crossover*, *recombination*, or *merge* operators. Some examples of this operation perform straightforward operations. Other examples involve more sophisticated strategies, in which, perhaps, two or more parent candidates are the inputs to a self-contained algorithmic search process which outputs one or more new 'child' candidates.

Finally, a randomized but competitive strategy is used to decide who will be a parent. The strategy is usually that the fitter a candidate is, the more likely it is to be selected to be a parent, and thus be the basis of a new candidate solution.

The existing, successful methods in approximate iterative optimization fall into two broad classes: *local search*, and *population-based search*. In *local search*, a special 'current' solution is maintained, and its neighbors are explored to find better quality solutions. Occasionally, one of these neighbors becomes the new current solution, and then its neighbors are explored, and so forth. The simplest example of a local search method is called hillclimbing. *Figure* 1.2 illustrates it.

Hillclimbing essentially iterates Idea 1 by continually producing new candidate solutions which are slight variations of the current solution. It also encapsulates a simple but effective variation on Idea 3: the solution which forms the basis for generating new candidates is always the best solution found so far, and a solution whose fitness is the best found so far. In simulated annealing, for example, the

difference is basically in step 3: sometimes, we will accept S' as the new current solution, even if it is worse then S. Without this facility, hillclimbing is prone to getting stuck at local peak solutions whose neighbors are all worse than the current solution where there is considerable scope for movement between solutions of equal goodness, but where very few or none of these solutions has a fitter neighbor. With this facility, simulated annealing and other local search methods can sometimes avoid such problems.

1. **Begin:** Generate and evaluate an initial 'current' solution S.
2. **Operate** Change S, produce S' and evaluate S'.
3. **Renew:** If S' is better than S, then overwrite S with S'.
4. **Iterate:** Unless a termination criterion is met, return to 2.

Figure 1.2: The Simple Hillclimbing Algorithm.

In *population-based search*, the notion of a single current solution is replaced by the maintenance of a population of different current solutions. New solutions are generated by first selecting members of this population to be parents, and then making changes to these parents to produce children. In population-based search, Idea 2 comes into play. Since there is now a collection of current solutions, rather than just one, we can exploit by using two or more from this collection at once as the basis for generating new candidates. Also, the introduction of a population brings with it further opportunities and issues. For example, Idea 3 is now open to more interesting realizations in which we use some strategy or other to select which solutions will be parents. Also, when one or more new candidate solutions have been produced, we need a strategy for maintaining the population. This is, assuming that we wish to maintain the population at a constant size, some of the population must be discarded in order to produce some or all of the new candidates via mutation or recombination operations. Notice, in particular, that a selection strategy combined with a population maintenance strategy adds up to an overall strategy for implementing a variation on Idea 3. Whereas the selection strategy encompasses the details of how to choose candidates from the population itself, the population maintenance strategy affects what is present in the population, and what therefore is selectable.

There are many *population-based* optimization algorithms and various ways to handle the above issues. The key characteristic of population-based algorithm is the variety of ways that have been used to implement Ideas 1, 2 and 3. Indeed, we tend to see a considerable degree of problem dependence; that is, the method which works best on a particular optimization problem may well not work best on another,

even closely related, problem. One consequence of this general observation that is of particular interest to us here is that it encourages research for new methods.

Continued investigation into local search and population based optimization techniques, and a consequent growing regard for their ability to address realistic problems well, has inspired much investigation of new and novel optimization methods. In this thesis, a special emphasis has been given to *Memetic Algorithms*, which represents one of the more successful emerging ideas in the ongoing research effort to understand population based and local search algorithms. The idea itself is to hybridize the two types of methods, and Memetic Algorithms represent a particular way of achieving the hybridization, and one which has achieved a considerable number of successes in recent years [CDG99].

1.4 Thesis Outline

We now give a summary of the following chapters of this thesis:

- **Chapter 2:** This chapter presents a general description of *Memetic Algorithms*. we also describe some fundamental concepts for further analysis and design, supply a description of the Local search and Population-based search, discuss some operators such as move, recombination, and finally give a general template for *Memetic Algorithms*.
- **Chapter 3:** This chapter provides an object-oriented framework for designing and implementing the template for *Memetic Algorithms*, which was introduceded in Chapter 2. A more detailed framework architecture is provided which includes Data classes, Manager classes, Runner classes, and Solver classes. Also a list of functions in the framework architecture which are classified into MustDef, MayReDef, and NonReDef are supplied.
- **Chapter 4:** In this chapter, as examples of actual applications of the framework, we develop the framework of *Memetic Algorithms* for two well-known combinatorial optimization problems, one is **Vertex Cover**, the other is **Independent Set**. We give a more detailed description about how to instantiate the object-oriented framework into an application program for a particular problem, and supply some detailed code which is on C++.
- **Chapter 5:** In this chapter, we will do testing on some graphs using the applications developed for *Vertex Cover* and *Independent Set*, and supply some testing results. These graphs include some random graphs, some benchmark graphs, and Papadimitriou and Steiglitz's Regular Graph which is special for *Vertex Cover*.
- **Chapter 6:** This chapter gives some conclusions of the thesis.

Chapter 2

Memetic Algorithms

2.1 General Description

Memetic Algorithms (MAs) are evolutionary algorithms (EAs) that apply a separate local search process to refine individuals (i.e. improve their fitness by hill-climbing). These methods are inspired by models of adaptation in natural systems that combine evolutionary adaptation of populations of individuals with individual learning within a lifetime. Additionally, MAs are inspired by Richard Dawkin's concept of a meme [Daw76], which represents a unit of cultural evolution that can exhibit local refinement. Under different contexts and situations, MAs are also known as hybrid EAs, genetic local searchers, Baldwinian EAs, Lamarkian EAs, etc.

MAs include a broad class of *metaheuristics*. This method is based on a population of agents and proved to be of practical success in a variety of problem domains. We can be sure that MAs constitute one of the most successful approaches for combinatorial optimization in general, and for the approximate solution of NP Optimization problems in particular.

Unlike traditional Evolutionary Computation approaches, MAs are concerned with exploiting all available knowledge about the problem under study. This is not as an optional mechanism, but as a fundamental feature.

From an optimization point of view, MAs are hybrid EAs that combine global and local search by using an EA to perform exploration while the local search method performs exploitation. Combining global and local search is a strategy used by many successful global optimization approaches, and MAs have in fact been recognized as a powerful algorithmic paradigm for evolutionary computing. In particular, the relative advantage of MAs over EAs is quite consistent on complex search spaces.

The characterization of a meme suggests that in culture evolution processes, information is not simply transmitting unaltered between individuals. In contrast, it is processed and enhanced by the communicating parts. This characteristic is accomplished in MAs by incorporating heuristics, approximate algorithms, local search techniques, specialized recombination operators, truncated exact methods, etc. Basically, most MAs can be regarded as a search strategy in which a population of optimizing agents cooperate and compete with each other [Mos01].

2.2 Memetic Algorithms

In this subsection, we will discuss and provide some basic concepts and definitions. These basic concepts will give rise to the notions of *local search* and *population-based search*, which MAs are founded on. The *population-based search* deals with recombination, a crucial mechanism in the functioning of MAs that will be studied later.

2.2.1 Some Fundamental Concepts

A computational problem P denotes a class of tasks and it has an input domain set of instances denoted I_P. For each instance $x \in I_P$, there is an associated set $sol_P(x)$ which denotes the feasible solutions for problem P given instance x. The set $sol_P(x)$ is also known as the set of acceptable or valid solutions.

We supply an algorithm to solve the problem P. Given instance $x \in I_P$, the algorithm should supply at least one element y from a set of *answers* $ans_P(x)$ that satisfies the requirements of the problem. In many cases, it may define a Boolean feasibility function $feasible_P(x,y)$ in order to identify whether a given solution $y \in ans_P(x)$ is acceptable for an instance $x \in I_P$ of a computational problem P.

In this thesis, we will focus on finding a solution in $sol_P(x)$ maximizing or minimizing a given function, i.e., optimization problem which will be solved by finding a certain feasible solution, either finding an optimal $y \in sol_P(x)$ or indicate that no such feasible solution exists. An algorithm is said to solve problem P if it can fulfill this condition for any given instance $x \in I_P$.

Here, we need a more restrictive characterization for our problems. This characterization is provided by restricting ourselves to the so-called combinatorial optimization problems. These combinatorial optimization problems constitute a special subclass of computational problems in which for each instance $x \in I_P$, it will satisfy:

- the cardinality of $sol_P(x)$ is finite.

- each solution $y \in sol_P(x)$ has *a goodness integer value* $m_P(y,x)$, obtained by an associated *objective function* m_P.
- a partial order $\prec_P$ is defined over the set of goodness values returned by the objective function, allowing determining which of two goodness values is preferable.

An instance $x \in I_P$ of a combinatorial optimization problem P is solved by finding the best solution $y^* \in sol_P(x)$, i.e., finding a solution y^* such that no other solution $y \prec_P y^*$ exists if $sol_P(x)$ is not empty. It is very common to have $\prec_P$ defining a total order. In this case, the best solution is the one that maximizes or minimizes the objective function [Mos01].

2.2.2 Search Landscapes

The notion of fitness landscapes has been introduced to describe evolutionary adaptation in nature and has become a powerful concept in evolutionary theory. Fitness landscapes are suitable for describing the behavior of heuristic search methods in optimization theory, since the process of evolution can be thought of as searching a collection of genotypes in order to find the genotype of an organism with highest fitness.

Regarding a heuristic search method as a strategy to navigate in the fitness landscape of a given optimization problem may be helpful for predicting the performance of a heuristic search algorithm. Furthermore, the analysis of a fitness landscape may help in designing highly effective search algorithm.

Before discussing search algorithms, three entities need to be discussed. These are the *search space*, the *neighborhood relation*, and the *guiding function*. For any given computational problem, these three entities can be instantiated in several ways, then that will give rise to different optimization tasks.

First of all, let us define the concept of *search space* for a combinatorial problem P. For this purpose, we consider a set $S_p(x)$, whose elements have the following properties:

- Each element $s \in S_p(x)$ represents at least one answer in $ans_p(x)$.
- For decision problems: at least one element of $sol_p(x)$ that stands for a 'Yes' answer must be represented by one element in $S_p(x)$.
- For optimization problems: at least one optimal element y^* of $sol_p(x)$ is represented by one element in $S_p(x)$.

Each element of $S_p(x)$ will be termed as a $configuration$, which is related to an answer in $ans_p(x)$ by a *growth function* g: $S_p(x) \rightarrow ans_p(x)$. Note that the first requirement refers to $ans_p(x)$, not to $sol_p(x)$, as some configurations in the search space may correspond to infeasible solutions. Thus the search algorithm may need to deal with this situation. If these requirements have been satisfied, it means that we have a valid representation or valid formulation of the problem. People who use biologically-inspired metaphors prefer to call $S_p(x)$ the *genotype space* and $ans_p(x)$ denotes the *phenotype* space, so we refer to g as the *growth function*.

To define a fitness landscape for a given problem instance, a real valued fitness has to be assigned to each of the solutions of the search landscape. Furthermore, we need to find an arrangement of the solutions or genotypes in the genotypical space to form a landscape.

The role of the search space is to provide a place on which the search algorithm will act. Important properties of the search space that affect the dynamics of the search algorithm are related to the accessibility relationships between the configurations. These relationships are dependent on a *neighborhood function* N. This function assigns to each element $s \in S$ a set $N(s) \subseteq S$ of neighboring configurations of s. The set $N(s)$ is called the *neighborhood* of s and each member s' $\in N(s)$ is called a *neighbor* of s.

It must be noted that the neighborhood depends on the instances, so the notation $N(s)$ is a simplified form of $N_p(s,x)$ since it is clear from the context. The elements of $N(s)$ need not be listed explicitly. In fact, it is very usual to define them implicitly by referring to a set of possible *moves*, which define *transitions* between configurations. Moves are usually defined as "*local*" modifications of some part of s, where locality refers to the fact that the move is done on a single solution to obtain another single solution. This locality is one of the key ingredients of *local search*, and actually it has also given the name to the whole search paradigm.

The last entity that must be defined is the *guiding function*. For this purpose, we require a set $\mathcal{F}$ whose elements are termed $fitness$ values and a partial order $\prec_f$ on $\mathcal{F}$. The guiding function is defined as a function F_g: $S \rightarrow \mathcal{F}$ that associates with each configuration $s \in S$ a value $F_g(s)$ that assess the quality of the solution. The behavior of the search algorithm will be controlled by these fitness values.

Notice that for optimization problems there is an obvious direct connection between the guiding function F_g and the objective function m_p. In fact, it is very common to enforce this relationship to the point that both terms are usually considered equivalent. However, this equivalence is not necessary and, in many cases, not even desirable. For decision problems, since a solution is a 'Yes' or 'No' answer, associated guiding functions usually take the form of distance to satisfaction.

The above difference between *objective function* and *guiding function* is also very important in the context of constrained optimization problems, i.e.,problems

for which in general $sol_p(x)$ is a proper subset of $ans_p(x)$. Since the growth function establishes a mapping from S to $ans_p(x)$, the search algorithm might need to process both feasible solutions and infeasible solutions. In many inplementations of Memetic Algorithms for these problems, a guiding function is defined as a weighted sum of the value of the objective function and the distance to feasibility which accounts for the constraints. Typically, a higher weight is assigned to the constraints, so as to give preference to feasibility over optimality.

The combination of a certain problem instance and the three entities defined above induce a so-called *fitness landscape*. Essentially, a fitness landscape can be defined as a directed graph, in which the vertices are configurations of the search space S, and the arcs connect neighboring configurations. The search can thus be seen as the process of 'navigating' the fitness landscape using the information provided by the guiding function. This is a very powerful and interesting metaphor; it may define some topographical objects such as *peaks*, *valleys*, *mesas*, etc, to simulate the search progress. In particular, a local optimum is regarded as a vertex of the fitness landscape whose guiding function value is better than the values of all its neighbors. We should also notice that different moves define different neighborhoods and hence different fitness landscapes, even when the same problem instance is considered.

Deep understanding of the idea and concept of the search landscape and its relevant techniques will be helpful to analyze combinatorial optimization problems and design highly effective search algorithms with special emphasis on Memetic Algorithms [Mos01].

2.2.3 Local and Population-Based Search

As we mentioned before, the existing, successful methods in approximate iterative optimization fall into two broad classes: *local search*, and *population-based search*, and most implementations of Memetic Algorithms include local search methods, it should firstly discuss *local search algorithm*. A local search algorithm starts from a configuration $s_0 \in S$, generated at random or constructed by some other algorithm. Subsequently, it iterates using at each step a transition based on the neighborhood of the current configuration. Transitions leading to preferable configurations are accepted, i.e., the newly generated configuration becomes the current configuration in the next step. Otherwise, the current configuration is kept. This process is repeated until a certain termination criterion is met.

Typical termination criteria are the realization of a pre-defined number of iterations, not having found any improvement in the last m iterations, or even more complex mechanisms based on estimating the probability of being at a local optimum.

Due to these characteristics, the approach is normally called "hill climbing". A Local Search Algorithm is sketched in *Fig.* 2.1.

```
Procedure Local-Search-Engine(current)
begin
    repeat
        new ← GenerateNeighbor(current);
        if (F_g(new) ≺_f F_g(current)) return
            current ← new;
        endif
    until TerminationCriterion();
    return current
end
```

Figure 2.1: A Local Search Algorithm.

The selection of the particular type of move (also known as *mutation* in the context of Genetic Algorithms) to use certainly does depend on the specific characteristics of the problem and the representation chosen. There is no common idea and suggestion. Finding a suitable encoding for the solutions of the problem is tightly linked with finding a good move operator for local search, since the move operator depends on the representation of the problem. The move operator defines the landscapes and thus is responsible for the performance of the neighborhood based search. Generally speaking, the move operator should only change the elements or genes of a solution slightly, since this often results in a slight change of fitness. However, in some cases, it is necessary to change many elements or genes to achieve a change in the fitness values.

The local search algorithm is characterized by keeping a single configuration at a time. Another kind of search strategy is to maintain more than one configuration at a time. The term *population-based* search algorithm has been introduced to denote this kind of search technique.

The availability of several configurations at a time allows the use of new powerful mechanisms for traversing the fitness landscape in addition to the standard mutation operator. The most popular of these mechanisms, the recombination operator, will be studied in more depth in the next section. We must notice that the general functioning of population-based search techniques is very similar to the pseudocode listed in *Fig.* 2.1. In fact, a population-based algorithm can be imagined as a procedure in which we visit vertices of a hypergraph sequentially. Each vertex of the hypergraph represents a set of configurations in $S_p(x)$, i.e., population. The next vertex to be visited, i.e., the population, can be established

according to the composition of the neighborhoods of the different transition mechanisms used in the population algorithm. So, in the next subsection, a new kind of operator which is based on more than one configuration will be supplied and discussed [Mos01].

2.2.4 Recombination

Recombination is a binary operator. In genetic algorithms, recombination is performed in analogy to biology by crossing over two parent bit vectors at a randomly selected position. Another widely used recombination mechanism is known under the name *uniform crossover*. Uniform crossover is a more general form of recombination, and single-point or k-point crossover are special cases.

As mentioned in the previous section, local search is based on the application of a mutation operator to a single configuration. For the population search, it uses recombination operators. In essence, the recombination can be defined as a process in which a set of S_{par} of n configurations which are informally referred to as "parents" is manipulated to create a set $S_{desc} \subseteq sol_p(x)$ of m new configurations which is informally termed a "descendant". The creation of these descendants involves the identification and combination of features extracted from the parents.

At this point, it is possible to consider different properties that can be exhibited by the recombination operator. The first property, *respect*, represents the exploitative aspect of recombination. A recombination operator is said to be $respectful$, regarding a particular type of feature of the configurations, if, and only if, it generates descendants carrying all basic features common to all parents. Note that, if all parent configurations are identical, a respectful recombination operator is forced to return the configuration as a descendent. This property is termed *purity*, and can be achieved even when the recombination operator is not generally respectful.

The second property, *assortment*, represents the exploratory side of recombination. A recombination operator is said to be *properly assorting* if, and only if, it can generate descendants carrying any combination of compatible features taken from the parents. The assortment is said to be *weak* if it is necessary to perform several recombination with the offspring to achieve this effect.

Finally, *transmission* is a very important property that captures the classical role of recombination. An operator is said to be transmitting if every feature exhibited by the offspring is present in at least one of the parents. Thus, a transmitting recombination operator combines the information present in the parents but does not introduce new information. The task of introducing new information is often left to the mutation operator.

The three properties above suffice to describe the abstract input and output behavior of a recombination operator regarding some particular features. It provides a characterization of the possible descendants that can be produced by the

operator. Nevertheless, there exist other aspects of the functioning of recombination that must be studied, in particular, how to approach the construction of S_{desc}.

A recombination operator is said to be *blind* if it does not use any information from the problem instance. So, the use of blind recombination operators has been usually justified because it does not introduce excessive bias in the search algorithm, and thus prevents extremely fast convergence to suboptimal solutions. However this is questionable. First, the behaviour of the algorithm is in fact biased by the choice of representation and the mechanics of the particular operators. Second, there exist widely known mechanisms to hinder these problems. Finally, it can be better to quickly obtain a suboptimal solution and restart the algorithm than use blind operators for a long time to get an asymptotically optimal behaviour.

Recombination operators that use problem knowledge are commonly termed heuristic or hybrid. In these operators, problem information is used to guide the process of constructing the descendants. This can be done in many ways for each problem, so it is difficult to provide a taxonomy of heuristic recombination operators. Nevertheless, there are two main aspects about which problem knowledge can be considered: the selection of the parent features that will be transmitted to the descendant, and the selection of the non-parental features that will be added to it. A heuristic recombination operator can focus on one of these aspects, or on both of them simultaneously.

Until now, we have supplied some basic concepts, described the idea of the fitness landscapes, briefly discussed the *local* and *population-based* search, and features of move operator and recombination operator. Based on the above mentioned, we will design and provide a general template for a *Memetic Algorithms* in the following subsection [Mos01].

2.3 Designing A Memetic Algorithm

According to the above considerations, it is possible to provide a general template for a memetic algorithm. As mentioned in Subsection 2.2.3, this template is very similar to that of a local search procedure acting on a set of $|pop| \geq 2$ configurations. This is shown in *Fig.* 2.2:

We will give some explanation for this template. First of all, the GenerateInitialPopulation procedure is responsible for creating the initial set of $|pop|$ configurations. This can be done by simply generating $|pop|$ random configurations or by using a more sophisticated seeding mechanism, by means of which high-quality configurations are created in the initial population [SR96] [LYY99]. For example, when solving the famous combinatorial optimization problem "Minimum Vertex Cover", we can use some greedy algorithm to supply initial configurations

for the population. Another possibility, the Local-Search-Engine presented in subsection 2.2.3 could be used as shown in *Fig.* 2.3:

As to the TerminationCriterion function, it can be defined very similarly to the case of Local Search, i.e., setting a limit on the total number of iterations, reaching a maximum number of iterations without improvement, or having performed a certain number of population restarts, etc.

```
Procedure Population-Based-Search-Engine
begin
    Initialize pop using GenerateInitialPopulation();
    repeat
        newpop ← GenerateNewPopulation(pop);
        pop ← UpdatePopulation(pop,newpop);
        if pop has converged then
            pop ←RestartPopulation(pop);
        endif
    until TerminationCriterion();
end
```

Figure 2.2: A Population-Based Search Algorithm.

```
Procedure GenerateInitialPopulation
begin
    Initialize pop using EmptyPopulation();
    parfor j ←1 to popsize do
        i ← GenerateRandomConfiguraion();
        j ← Local-Search-Engine(i);
        InsertInPopulation individual i to pop;
    endparfor
    return pop
end
```

Figure 2.3: Creating High-Quality Solutions in the Initial Population.

The GenerateNewPopulation procedure is the core of the memetic algorithm. Essentially, this procedure can be seen as a pipeline process comprising n_{op} stages. Each of these stages consists of taking $arity_{in}^{j}$ configurations from the previous stage, generating $arity_{out}^{j}$ new configurations by applying an operator op^{j}. This pipeline is restricted to have $arity_{in}^{1} = popsize$. The whole process is listed in *Fig.* 2.4:

```
Procedure GenerateNewPoplation(pop)
begin
    buffer^0 ← pop
    parfor j ←1 to n_op do
        Initialize buffer^j using EmptyPopulation();
    endparfor
    parfor j ←1 to n_op do
        S^j_par ← ExtractFromBuffer(buffer^{j-1}, arity^j_in);
        S^j_desc ← ApplyOperator(op^j, S^j_par);
        for z ←1 to arity^j_out do
            InsertInPopulation individual S^j_desc[z] to buffer^j;
        endfor
    endparfor
    return buffer^{n_op}
end
```

Figure 2.4: The GenerateNewPopulation procedure.

This template for the GenerateNewPopulation procedure is usually instantiated in Genetic Algorithms by letting $n_{op} = 3$, using a selection, a recombination, and a mutation operator. Traditionally, mutation is applied after recombination. However, if a heuristic recombination operator is being used, it may be more convenient to apply mutation before recombination. Since the purpose of mutation is simply to introduce new features into the configuration pool, using it in an advance is also possible. Furthermore, the smart feature combination performed by the heuristic operator would not be disturbed this way.

This situation has a little different pattern in Memetic Algorithms. In this case, it is very common to let $n_{op} = 5$, inserting a Local-Search-Engine right after applying op^2 and op^4 which are recombination and mutation respectively. Due to the local optimization performed after the mutation, applying the latter after recombination is not as problematic as in Genetic Algorithms.

The UpdatePopulation procedure is used to reconstruct the current population using the old population *pop* and the new generated population *newpop*. There are two main possibilities to carry out this reconstruction: the *plus* strategy and the *comma* strategy. In the former, the current population is constructed taking the best *popsize* configurations from $pop \cup newpop$. In the latter, the best *popsize* configurations are taken just from *newpop*, In this latter case, it is required to have $|newpop| > popsize$, so as to put some selective pressure on the process. Otherwise, the search would reduce to a random wandering through S.

There are a number of studies regarding appropriate choice for the UpdatePopulation procedure. As a general guideline, the comma strategy is usually regarded

as less prone to stagnation, being the ratio $|newpop|$ / $popsize \simeq 6$ a common choice. Nevertheless, this option can be somewhat computationally expensive if the guiding function is complex and time-consuming. Another common alternative is using a plus strategy with a low value of $|newpop|$. This option usually provides a faster convergence to suboptimal regions of the search space, i.e., all configurations in the population being very similar to each other, hence hindering the exploration of other regions of S.

The above consideration about premature convergence leads to the last component of the template shown in *Fig.* 2.2, the restarting procedure. First of all, it must be decided whether the population has, or has not, degraded . To do so, it is possible to use some measure of information diversity in the population such as Shannon's entropy [DBK92].

Once the population is considered to be at a degenerate state, the restart procedure will be invoked. Again, this can be implemented in a number of ways. A very typical strategy is keeping a fraction of the current population, (this fraction can be as small as one solution, the current best), and substituting the remainder of configurations with newly generated solutions, as shown in *Fig.* 2.5:

```
Procedure RestartPopulation(pop)
begin
    Intialize newpop using EmptyPopulation();
    #preserved ← popsize%preserve;
    for j ← 1 to #preserved do
        i ← ExtractBestFromPopulation(pop);
        InsertInPopulation individual i to newpop;
    endfor
    parfor j ← #preserved+1 to popsize do
        i ← GenerateRandomConfiguraion();
        j ← Local-Search-Engine(i);
        InsertInPopulation individual i to newpop;
    endparfor
    return newpop
end
```

Figure 2.5: The RestartPopulation Procedure.

The procedure shown in *Fig.* 2.5 is also known as the *random immigrant* strategy [CG93]. A different possibility is activating a *strong* or *heavy* mutation operator in order to drive the population away from its current location in the search

space. Both options have their advantages and disadvantages. For example, when using the random immigrant strategy, one has to take some caution to prevent the preserved configurations from taking over the population. As to the heavy mutation strategy, one has to achieve a tradeoff between an excessively strong mutation that would destroy any information contained in the current population, and a not strong enough mutation that would cause the population to converge again in a few iterations [Mos01].

Chapter 3

Framework

3.1 General Description

In the last chapter, a general template for a memetic algorithm was presented. We believe that the use of an object-oriented *framework* can be helpful in designing and implementing this template. A framework is a special kind of software library, which consists of a hierarchy of abstract classes. The user needs only to define suitable derived classes, which implement the virtual functions of the abstract classes. A Framework is characterized by the *inverse control* mechanism (also known as the *Hollywood Principle*: "Don't call me, we'll call you") for the communication with the user code: the functions of the framework call the user-defined ones and not the other way round. The framework thus provides the full control structures of the invariant parts of the algorithms, and the user only supplies the problem specific details. On the contrary, libraries that instead use a direct control mechanism, such as LEDA [MNU99], are called *toolkits* in the *object-oriented* jargon.

The basic idea for the development of an object-oriented application is the exploitation of already available *Design Patterns*. They are abstract structures of classes that are commonly presented in object-oriented applications and frameworks that have been precisely identified and classified. Their use allows the designer to address many implementation issues in a more principled way.

In this thesis, an object-oriented framework was present to be used as a general tool for the development and the implementation of *Memetic Algorithms* in *C++*. The basic idea of framework is to capture the essential features of most memetic algorithm techniques, and their possible compositions.

The framework relies on two design patterns [GHJV94], namely the *template method*, to specify and implement the invariant parts of various search algorithms, and the *strategy method*, for the communication between the main solver and its component classes.

Furthermore, the framework provides a principled modularization for the design of *Memetic Algorithms*, and it exhibits several advantages with respect to directly implementing the algorithm from scratch not only in terms of reuse, but also in methodology and conceptual clarity. Moreover, the framework is easily extensible by means of new class derivation and composition. The above features mitigate some potential drawbacks of the framework, such as the loss of flexibility and the computational overhead in the implementation of the algorithm.

3.2 Framework Architecture

The core of the framework is composed by a set of cooperating classes that take care of different aspects of the template of *Memetic Algorithms*. The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithm. In fact, the relationships between classes, and their interactions by mutual method invocations, are completely dealt with by the framework.

The classes in the framework are split into four categories:

- **Data Classes:** store the basic data of the algorithm. They maintain the states of the *search space*, the *move*, the *input*, and the *output* data. These classes have only data members and no methods, except for those accessing their own data. They have no computing capabilities and no link to other classes.

- **Managers:** perform actions related to some specific aspects of the search. For example, the Neighborhood Manager is responsible for everything about the neighborhood: candidate move selection, update current states by executing some moves, and so on. Different Neighborhood Managers may be defined in case of different searches, each one handling a specific neighborhood relation used by the algorithms. Managers do not have their own internal data, but they work on the internal state of the runners that invoke them, and interact with the internal state through function parameters.

- **Runners:** are responsible for performing a run of a local search technique, starting from one initial individual state in the population and leading to the final one. Each runner has many data objects for representing the state of the search, such as current state, current move, the number of iterations, etc., and maintain links to all the managers, which are invoked for performing specific tasks on its own data. The example of runners is Hill Climbing search.

- **Solvers:** control the search by generating the initial solutions, and deciding how and in which sequence the search has to be activated. In addition, the solvers communicate with the user, by getting the input and delivering the output. They represent the most external software layer of the framework, and are linked to one or more runner and to some of the managers.

The main classes that comprise the current version of the framework are depicted in *Fig* 3.1 in UML notation [BRJ99]. The data classes, shown in small boxes, are supplied to the other classes as templates, which need to be instantiated by the user with the corresponding problem-specific types. Classes whose names are in normal font represent the *interface* of the framework with respect to the user, and they are meant for direct derivation of the user's concrete classes. Conversely, classes in italic typeface are used only as base classes for the other framework classes.

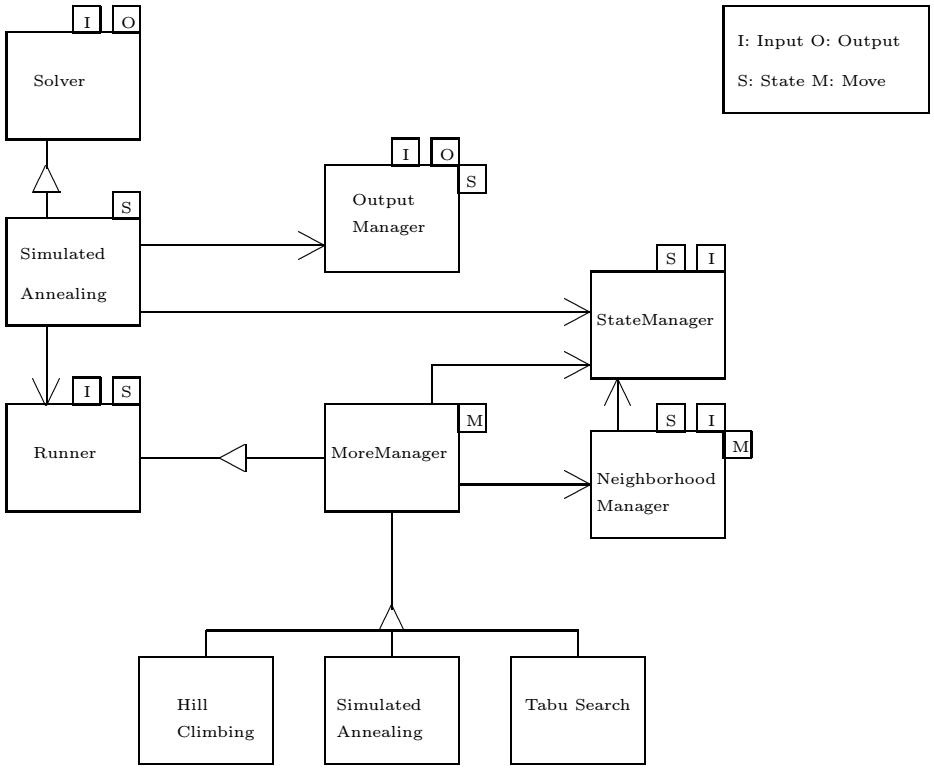

Figure 3.1: Framework Main Classes

In *Fig*. 3.1, templates that are shared by a hierarchy of classes are shown only on the base class. For example, the class Hill Climbing inherits the three templates *Input*, *State*, and *Move*.

Notice that the use of template classes for input and output forces the client to define two specific classes for dealing with the input and the output of the search procedure. This is a deliberate design that encourages the client to identify explicitly input and output objects.

Methods of framework interface classes are in turn split in three categories that we call *MustDef, MayReDef, NoReDef* functions.

- **MustDef:** *pure virtual C++* functions which correspond to the problem specific aspects of the algorithm; they must be defined by the user.
- **MayReDef:** *non-pure virtual C++* functions and they come with a tentative definition, which may be redefined by the user in case the definition is not satisfactory for the problem at hand. Because of the *late binding* mechanism for virtual functions, the program always invokes the user-defined version of the function.
- **NoReDef:** *final(non-virtual), non-virtual C++* functions which cannot be redefined by the user. More precisely, they can be redefined, but the base class version is executed when invoked through the framework.

In order to use framework, the user has to define the data classes (i.e., the template instantiations), the derived classes for the managers, and at least one runner and one solver.

Many of the framework classes have no $MustDef$ functions, thus the corresponding subclass to be defined by the user comprises only the class constructor, which in $C++$ cannot be inherited. For all user classes, the framework provides a skeleton version, which is usually suitable for the user application. The skeleton comprises the definition of the classes, the declaration of the constructors, the $MustDef$ functions and all the necessary include directives. The user thus has only to fill in the empty $MustDef$ functions. Therefore, as discussed in later, very little code needs to be actually written by the user.

3.3 Framework Main Components

We now describe in more details the classes that comprise the framework. We thus present data classes, managers, runners, and solvers, together with their interaction.

3.3.1 Data Classes

The data classes are templates, and therefore they have no actual code. They serve for storing the following data (the example refers to Vertex Cover problem):

- **Input:** input data of the problem: e.g. the size of population.
- **Output:** output to be delivered to the user; e.g. all states of vertex covers in the population.
- **State:** an element of the search space; e.g. the vertex cover V'.
- **Move:** a local move; e.g. a vector of integer representing the change of vertex cover node.

In a few applications, State and Output classes may coincide. In general, however, the *output space* that is related to the problem specification and the *search space* that regards the algorithm are different.

3.3.2 Manager Classes

The framework provides three manager classes. Managers are not related hierarchically, but they are linked to runners and to each other through pointers. The managers are as follows.

- **StateManager:** responsible for all operations on the individual state that are independent of the neighborhood definition, such as generating a random state, and computing the cost of a state. No move definition is supplied to the StateManager.
- **OutputManager:** responsible for translating between elements of the search space and output solutions. It also delivers other output information of the search, and stores and retrieves solutions from files. This is the only manager that deals with the Output class. All other managers work only on the State class, which represents the elements of the search space used by the algorithms.
- **NeighborhoodManager:** (detail will be shown later)

Next, all managers will be discussed in more detail.

NeighborhoodManager

The NeighborhoodManager is associated to a specific *Move* class: therefore, if different neighborhood relations are used, different subclasses of NeighborhoodManager have to be defined with different instantiations for the template Move.

Some of the main functions of the NeighborhoodManager are the following:

MustDef functions:

- **MakeMove(State &st, const Move &mv):** updates the state *st* by applying the move *mv* to it.
- **DeltaObjective(const State &st, const Move &mv):** computes the difference in the objective function between the state obtained from *st* applying *mv* and the state *st* itself.
- **RandomMove(const State &st, Move &mv):** generate a random move for the state *st* and store it in *mv*.
- **NextMove(const State &st, Move &mv):** modifies *mv* to become the candidate move that follows *mv* according to the neighborhood exploration strategy. This is used in algorithms relying on exhaustive neighborhood exploration.

MayReDef functions:

- **FirstMove(const State &st, Move &mv):** generates the first move for the state *st* according to the neighborhood exploration strategy, and store it in *mv*. Its tentative definition simply invokes the *RandomMove* method.
- **BestMove(const State &st, Move &mv):** computes the best possible move in the neighborhood of *st*.
- **SampleMove(const State &st, Move &mv ,int n):** computes the best one among *n* random neighbors of *st*.

As an example of the framework code, we present the definition of *BestMove*, Here fvalve denotes the codemain of the object function (typically int or double), and LastMoveDone() is a MayReDef function whose tentative code is the single instrution "return mv == start_move;".

```
template<typename Input, typename State, typename Move>
fvalue NeighborhoodManager <Input, State, Move>
    ::BestMove(const State &st, Move &mv)
{
```

```
    FirstMove(st,mv);
    fvalue mv_cost = DeltaObjective(st, mv);
    best_move = mv;
    start_move = mv;
    fvalue best_delta = mv_cost;

    do
    {
        m_cost = DeltaObjective(st, mv);
        if(mv_best < best_delta)
        {
            best_move = mv;
            bes_delta = mv_cost;
        }
        NextMove(st,mv);
    }while(!LastMoveDone(mv))
    mv = best_move;
    return best_delta;
}
```

The NeighborhoodManager also can include functions for more sophisticated selection mechanisms according to the search strategy. For example, the function EliteMove() selects an improving move from a list of elite candidates. The list is rebuilt from scratch whenever all its members are non-improving in the current state.

OutputManager

OutputManager is responsible for translating between elements of the search space and output solutions. It also delivers other output information of the search, and stores and retrieves solution from files.

This is the only manager that deals with the Output class. All other managers work only on the State class, which represents the elements of the search space used by the algorithms.

Its main functions are the following ones. The most used one is OutputState() which delivers the output at the end of the search.

MustDef functions:

- **InputState(vector<State> &pop, Output &out):** gets the state population *pop* from the output *out*.

- **OutputState(const vector<State> &pop, Output &out):** writes the output *out* from the state population *pop*.

MayReDef functions:

- **ReadState(vector<State> &pop, istream &is):** reads the state population *pop* from the stream *is* (it uses InputState()).
- **WriteState(const vector<State> &pop, ostream &os):** writes the state population *pop* in the stream *os* (it uses OutputState()).

StateManager

The StateManager is responsible for all operations on the states that are independent of the neighborhood definition. No move definition is supplied to the State Manager.

Its core functions are the following ones:

MustDef functions:

- **RandomState(State &st):** makes *st* become a random state.
- **Objective(const State &st):** computes the value of the objective function in the state *st*.

MayReDef functions:

- **SampleState(State &st, int n):** stores in *st* the best among n randomly generated states.
- **ImprovedSampleState(State &st, int n, Runner* r):** generates a state and stores it in *st*. The state is chosen as the best among the final states of n runs of the runner r on intial random states.

The idea of the function ImprovedSampleState() is to use a fast local search strategy (runner r) on a set of random states, for finding the initial state of the main (possibly slow) runner.

3.3.3 Runner Classes

The framework comprises a hierarchy of runners. The base class *Runner* has only *Input* and *State* templates, and it is connected to the solvers, which have no knowledge about the neighborhood relations.

The class *MoveRunner* requires also the template *Move*, and the pointers to the necessary Managers. It also stores the basic data common to all derived classes: the current state, the current move, and the number of iterations.

The use of templates allows us to define directly objects of type *State*, such as current state and best state, rather than accessing them through pointers. This makes construction and copy of objects of type State completely transparent to the user, not requiring any explicit cast operation or dynamic allocation.

Here, we present *Go*(), the main function of *MoveRunner*, which performs a full run of local search.

```
template<typename Input, typename State, typename Move>
void MoveRunner<Input, State, Move>::Go()
{
    InitializeRun();
    while (ContinueSearching() &&!LowerBoundReached())
    {
        UpdateIterationCounter();
        SelectMove();
        if(AcceptableMove())
        {
            MakeMove();
            StoreMove();
        }
    }
    TerminateRun();
}
```

Most of the functions invoked by Go() are defined in the subclasses. It depends on what kind of search strategy will be used in the local search, what kind of NeighborhoodManager will be pointed to.

For example, being *p_nhe* the pointer to the NeighborhoodManager, the SelectMove() function invokes $p\_nhe \rightarrow RandomMove()$ in the subclass HillClimbing, whilst in the subclass TabuSearch it invokes $p\_nhe \rightarrow BestNonProhibitedMove()$.

Two functions which are not defined below in the hierarchy are the MayRedef functions $UpdateIterationCounter()$ and $LowerBoundReached()$. Their tenta-

tive definition simply consists in incrementing the iteration counter by one, and checking if the current state cost is equal to 0, respectively.

3.3.4 Solver Classes

Solvers represent the external layer of the framework. Their code is almost completely provided by framework classes; i.e. they have no MustDef functions. Solvers have an internal state and pointers to one or more runners.

The main functions of a solver are the following ones.

MayReDef functions:

- **Run():** starts the memetic algorithm search process, it is the core part of the framework.
- **GenerateInitPopulation():** gets the initial state population by calling the function RandomState() of the StateManager on the internal state of the solver.

NoReDef functions:

- **DeliverOutput():** calls the function OutputState() of the OutputManager on the internal state population of the solver.
- **Solve():** makes a complete execution of the solver, by invoking the functions FindInitialState(), Run(), and DeliverOutput().

GenerateInitPopulation() function

The core of its (simplified) definition of GenerateInitPopulation() is given below.

```
template<typename Input, typename Output, typename State, typename Move>
void SolverForPopulationSearch<Input, Output, State, Move>
    ::GenerateInitPopulation()
{
    ...
    StateManager<Input, State> stateManager1(&input);
    NeighborhoodManager<Input, State, Move> neighborhoodManager1(&input);

    HillClimbing<Input1,State1,Move1>
        hillClimbing1(&input, &stateManager1, &neighborhoodManager1);

    int numberOfPop = input.getNumberOfPop();
```

```
    for( int i = 0; i < numberOfPop; ++i)
    {
        State1 state(numberOfNode);
        stateManager1.RandomState(state);

        hillClimbing1.InitializeRun();
        hillClimbing1.SetCurrentState(state);
        hillClimbing1.Go();

        state = hillClimbing1.GetCurrentState();
        pop.push_back(state);
    }
    ...
}
```

Note that both solvers and runners have their own state variables, and they communicate through the functions *GetCurrentState*() and *SetCurrentState*().

Run() function

Various solvers differ from each other mainly in the definition of the Run() function. For example, if the user wants to improve the memetic algorithm or use another kind of search algorithm, he needs to modify the Run function.

The core of its (simplified) definition is given below. The solver's variable internal state population is previously set to the initial state population by the function *GenerateInitPopulation*().

```
template<typename Input, typename Output, typename State, typename Move>
void SolverForPopulationSearch<Input, Output, State, Move>::Run()
{
    ...
    while(TerminationCriterion())
    {
        UpdateIterationCounter();
        newpop = GenerateNewPopulation(pop);
        UpdatePopulation(pop, newpop);

        if(Converged(pop))
            RestartPopulation(pop);
    }
    ...
```

```
}
```

The main part of the Run() is nearly coincidental with the core idea of memetic algorithm listed in subsection 2.3 “Designing A Memetic algorithm”.

DeliverOutput function

The core of its (simplified) definition of DeliverOutput() is given below.

```
template<typename Input, typename Output, typename State, typename Move>
void SolverForPopulationSearch<Input, Output, State, Move>
::DeliverOutput()
{
    ...
    OutputManager<Input, Output, State> outputManager1;

    outputManager1.OutputState(pop, output);
    output.printOutput();
    ...
}
```

It calls the function *OutputState*() of the *OutputManager* on the internal state population of the solver.

Chapter 4

Applications of the Framework for Memetic Algorithms

In the last chapter, the object-oriented $framework$ was described for designing and implementing the template of Memetic Algorithms. The core of the framework is composed of a set of cooperating classes that deal with different aspects of the template of Memetic Algorithms. The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithm, and the relationships between classes, and their interactions by mutual method invocation, are completely dealt with by the framework,

In this chapter, as examples of actual use of the framework, we present the development of the framework of the Memetic Algorithms for two well-known combinatorial optimization problems, one is **Vertex Cover**, the other is **Independent Set**.

First of all, several basic definitions and concepts in the theory of graphs and relevant algorithms about Vertex Cover and Independent Set need to be introduced. A few results involving these concepts will be established. These results, while illustrating the concepts, will also serve to introduce certain techniques commonly used in the algorithms that will be discussed later.

4.1 Basic Graph Definitions and Algorithms

4.1.1 Preliminary Definitions

Some standard definitions and notions for the following discussion and citation will be stated. All of these definitions in the theory of graphs are from [Din99].

Definition 1 *A* **graph** *G is a pair G = (V,E) consisting of a finite set V ≠ 0 and a set E of two-element subsets of V. The elements of V are called* **vertices**. *An element e = (a,b) of E is called an* **edge** *with end vertices a and b.*

We say that a and b are **incident** with e and that a and b are adjacent or neighbors of each other.

We often illustrate graphs by pictures in the plane. The vertices of a graph $G = (V,E)$ are represented by points and the edges by lines connecting the end points.

Definition 2 *A* **digraph** *is a pair G = (V,E) consisting of a finite set V and a set E of ordered pairs (a,b), where a ≠ b are elements of V. The elements of V are called* **vertices** *again, those of E edges are called* **arc** *here.*

The term **arc** is used to distinguish between the directed and the undirected case. Instead of $e = (a,b)$, we write $e = ab$. a is called the start vertex or tail and b the end vertex or head of e. An arc that begins and ends at the same vertex u is called a **loop**. By being defined as a set, E does not contain duplicate (or multiple) edges between the same vertices.

Definition 3 *The* **order** *of a graph G = (V,E) is |V|, sometimes denoted by |G|, and the size of this graph is |E|.*

Sometimes we view a graph as a digraph where every unordered edge *(u,v)* is replaced by two directed arcs *(u,v)* and *(v,u)*. In this case, the size of a graph is half the size of the corresponding digraph.

Definition 4 *A graph G is* **connected** *if there is a path between all pairs of vertices u and v of V(G). A digraph is* **strongly connected** *if there is a path from vertex u to vertex v for all pairs u and v in V(G).*

Definition 5 *In a graph, the* **degree** *of a vertex v, denoted by deg(v), is the number of edges incident to v. For digraphs, the out-degree of a vertex is the number of arcs (v,x) ∈ E | x ∈ V incident from v (leaving v) and the in-degree of vertex v is the number of arcs (x,v) ∈ E | x ∈ V incident to v (entering).*

Definition 6 *The* **distance** *between two vertices u and v in G, denoted by d(u,v), is the length of the shortest u - v path in G, If no such path exists, then we define d(u,v) to be infinite.*

For most graphs, there is likely at most one edge from any vertex v to any other vertex w (this allows one edge in each direction between v and w). Consequently, $|E| \leq |V|^2$. When most edges are present, we have $|\mathrm{E}| = \Theta(|\mathrm{V}|^2)$, and the graph is considered **dense**, that is, it has a large number of edges. In most applications, the graph is **sparse** rather than dense.

4.1.2 Computer Representations of Graphs

Assume that the vertices are sequentially numbered starting from 0. One simple way to represent a graph is to use a two-dimensional array. This is an **adjacency matrix** representation. For a graph G of order n, and adjacency matrix representation is a boolean matrix (often encodes with 0's and 1's) of dimension n. For each edge (v,w), we set $m[v][w]$ equal to 1; nonexistent edges can be set to 0. The adjacency matrix $M = [m_{ij}]$ of G is an n $\times$ n matrix with m_{ij} defined as follows:

$$m_{ij} = \begin{cases} 1, \text{ if } (v_i, v_j) \in E \\ 0, \text{ otherwise.} \end{cases}$$

For sparse graphs, a better solution is an **adjacency list** representation. For each vertex we keep a linked list of all adjacent vertices.

Because each edge appears in a list node, the number of list nodes is exactly equal to the number of edges. Consequently, $O(|E|)$ space is used to store the list nodes. Since we have $|V|$ lists, there is also $O(|V|)$ additional space that is required. If we assume that every vertex is in some edge, then the number of edges is at least $\lceil |V|/2 \rceil$. So we may disregard any $O(|V|)$ terms when an $O(|E|)$ term is present. Consequently, we say that the space requirement is $O(|V|)$, or linear in the size of the graph. However to check whether $edge/arc(i,j)$ is in the graph the adjacency matrix representation has constant-time lookup, while the adjacency lists representation may require $O(n)$ time in the worst case. We should point out that there are also other specialized graph representations besides the two mentioned in this section. These data structures take advantage of the graph structure for improved storage or access time, often for families of graphs sharing a common property.

4.2 The First Case Study: Vertex Cover

4.2.1 The Definition and Greedy Algorithm

The minimum **vertex cover** is a combinatorial optimization problem with many practical applications in, for example, computer networking and scheduling. Vertex cover formulated as a decision problem is *NP-Complete*.

Problem 7 *The* **Vertex Cover** *optimization problem is to find a cover of minimum size in the given graph, i.e., given undirected graph G consisting of nodes V and edges E, G = (V,E), find a minimum size subset of nodes Vc $\in$ V such that every edge in E is incident on at least one of the nodes in V_c. V_c is said to be a vertex cover of Graph G.*

Algorithm 8 (GVC) *The greedy heuristic algorithm for vertex cover is to randomly select the vertex with highest degree, add it to the cover, delete all adjacent edges, and then repeat until the graph is empty.*

Procedure GVC(Graph G)
begin
    Initialize $V_c \leftarrow \phi$;
    while the graph is not empty
        Randomly select one vertex v with highest degree;
        Append v to V_c;
        Delete all adjacent edges,
    endwhile
    return V_c
end

Figure 4.1: The Greedy Heuristic Algorithm for Vertex Cover.

The core of framework is composed by a set of cooperating classes that take care of different aspects of the template of memetic algorithms. As described in subsection 3.2 on page 20, these classes in the framework are split in four categories: *Data classes, Manager classes, Runner classes*, and *Solver Classes*, and methods of framework interface classes are in turn split in three categories that we call *MustDef, MayReDef, NoReDef* functions. we need to define $MustDef$ functions, and maybe redefine some $MayReDef$ function as demand.

The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithm. In the following subsection, the detail descrptions for instantiating these classes in the framework will be presented.

4.2.2 Data Classes

The data classes are templates, and therefore they have no actual code. They serve for storing the data, so, first of all, the data classes need to be instantiated.

In the **Input** class, it includes an graph which can be defined using the facilities of the *LEDA* library, the number of the state in the population, some number of the iteration, and other relevant data, such as the update Strategy of the population in the memetic algorithm.

In the **Output** class, it includes an integer vector representing the vertex cover and other relevant data.

In the **State** class, it includes an integer vector representing the vertex cover.

In the **Move** class, it includes an integer vector representing the vertex cover status changing of a node in the graph.

4.2.3 Manager Classes

According to the detail description in subsection 3.3.2, the framework provides three manager classes, **StateManager, NeighborhoodManager, and OutputManager**, Manager classes are not related hierarchically, but they are linked to runners and to each other through pointers.

StateManager Class

The **StateManager** class is responsible for all operations on the individual state that are independent of the neighborhood definition, such as generating a random state, and computing the cost of a state. No move definition is supplied to the StateManager.

In the StateManager class in the application for Vertex Cover, **RandomState** function and **Objective** function need to be defined.

RandomState function normally is for generating an initial state randomly. It is $MustDef$ function.

For the problem of Vertex Cover, *Greedy Algorithm* 8 in subsection 4.2.1 is used for generating individual initial state. The core part of code follows:

```
template<typename Input, typename State>
void StateManager<Input, State>::RandomState(State &state)
{
    leda_node v;
    leda_edge e;
    vector<leda_node> vectorOfNodes;
    vector<leda_edge> vectorOfEdges;
    leda_graph G = input→getGraph();

    forall_nodes(v,G) vectorOfNodes.push_back(v);
    forall_edges(e,G) vectorOfEdges.push_back(e);

    int count0 = 0;
    vector<int> tem = state.getValue();
    int numberOfNode = input→getNumberOfNode();
```

```
    int numberOfCurrentNode = 0;
    int numberOfEdges = 0;

    do{
        vector<int> vectorOfEdgeNumber;
        vector<int> vectorOfPosition;
        forall_nodes(v,G)
        {
            count0=0;
            forall_inout_edges(e,v) count0++;
            vectorOfEdgeNumber.push_back(count0);
        }

        count0=vectorOfEdgeNumber[0];
        for(int i = 1; i < vectorOfEdgeNumber.size();i++)
        {
            if(count0<vectorOfEdgeNumber[i])
                count0 = vectorOfEdgeNumber[i];
        }
        for(int i = 1; i < vectorOfEdgeNumber.size();i++)
        {
            if(count0==vectorOfEdgeNumber[i])
                vectorOfPosition.push_back(i);
        }
        do{
            numberOfCurrentNode =
                vectorOfPosition[int((double(rand())/RAND_MAX)
                    *vectorOfPosition.size())];
            count0 = 0;
            forall_inout_edges(e,vectorOfNodes[numberOfCurrentNode]) count0++;
        }while(tem[numberOfCurrentNode]!=0 || count0 ==0);
        if(count0 > 0)
        {
        tem[numberOfCurrentNode] = 1;
        forall_inout_edges(e,vectorOfNodes[numberOfCurrentNode]) G.del_edge(e);
        }
    }while(G.number_of_edges());
    state.setValue(tem);
}
```

In the framework, the **Objective** function normally is for computing the value of the objective function in the state *st*. In the encoding of the problem of Vertex

Cover, the approach for representing the state *st* is a direct encoding in which each bit of a binary array of length |V| defines the presence or absence of the corresponding vertex in the cover. The objective() function returns the number of bits in the state which are true.

The core part of code is following:

```
template<typename Input, typename State>
void SolverForPopulationSearch<Input, State >::Objective(State &state)
{
    vector<int> tem = state.getValue();
    int temObjective = 0;

    for(int i = 0; i < tem.size(); i++)
    {
        if(tem[i] == 1) temObjective++;
    }
    return temObjective;
}
```

NeighborhoodManager Class

The **NeighborhoodManager** class is associated to a specific *Move* data class: therefore, if different neighborhood relations are used, different subclasses of NeighborhoodManager have to be defined with different instantiations for the template Move.

In the NeighborhoodManager class in the application for Vertex Cover, **MakeMove** function and **RandomMove** function need to be defined. They are all *MustDef* function.

MakeMove function is for updating the state *st* by applying the move *mv* to it, the core part of code is following:

```
template<typename Input, typename State, typename Move>
void NeighborhoodManager<Input, State, Move>
    ::MakeMove(State &state, Move &move)
{
    vector<int> tempForState = state.getValue();
    vector<int> tempForMove = move.getValue();

    for(int i = 0; i < tempForState.size(); i++)
        tempForState[i] = (tempForState[i]+tempForMove[i])%2;
```

```
    state.setValue(tempForState);
}
```

RandomMove function is for generating a random move for the state *st* and store it in *mv*, the core part of code is following:

```
template<typename Input, typename State, typename Move>
void NeighborhoodManager<Input, State, Move>
    ::RandomMove(const State &state, Move &move)
{
    vector<int> delta;
    int numberOfNode = input→getNumberOfNode();

    for(int i = 0; i <numberOfNode; i++)
        delta.push_back(int((double(rand())/RAND_MAX)*2));

    move.setValue(delta);
}
```

OutputManager Class

The **OutputManager** class is responsible for translating between elements of the search space and output solutions. It also delivers other output information of the search, and stores and retrieves solution from files.

In the OutputManager class, we need to define **OutputState** function and other functions.

For the **OutputState** function, the core part of code is following:

```
template<typename Input, typename Output, typename State>
void OutputManager<Input, State, Move>
    ::OutputState(const vector<State> pop, Output &output)
{
    output.setValue(pop);
}
```

4.2.4 Runner Classes

Runner classes are responsible for performing a run of a *local search* technique, starting from one initial individual state in the population and leading to the final one.

Each runner has many data objects for representing the state of the search, such as current state, current move, the number of iterations, etc., and maintains links to all the managers, which are invoked for performing specific tasks on its own data. One example of the runners is hill climbing search.

The framework comprises a hierarchy of runners. The base class *Runner* has only *Input* and *State* templates, and it is connected to the solvers, which have no knowledge about the neighborhood relations.

The class *MoveRunner* requires also the template *Move*, and the pointers to the necessary managers. It also stores the basic data common to all derived classes: the current state, the current move, and the number of iterations.

In the subsection 3.3.3, we present the main code of *Go()*, the main function of MoveRunner, which performs a full run of *local search*.

Most of the functions invoked by *Go()* are defined in the subclasses of *MoveRunner*. It depends on what kind of search strategy will be used in the local search, what kind of *NeighborhoodManager* will be pointed to.

Here we define a subclass of MoveRunner, **HillClimbing**.

```
template<typename Input, typename State, typename Move>
void NeighborhoodManager<Input, State, Move>
    class HillClimbing : public MoveRunner<Input, State, Move>
{
    ...

    HillClimbing(Input *input,
        StateManager<Input, State> *stateManager,
        NeighborhoodManager<Input, State, Move> *neighborhoodManager)
        : MoveRunner<Input, State, Move>(input, stateManager, neighborhoodManager)
        { }

    ...

    protected:
    void SelectMove()
    {
    neighborhoodManager1→RandomMove(state, move);
    }
```

```
    void MakeMove()
    {
    neighborhoodManager1→MakeMove(state, move);
    }
    ...
}
```

In **SelectMove()** function, *RandomMove* neighborhood relationship which is supplied by the NeighborhoodManager is invoked. In **MakeMove()** function, *MakeMove* function which is also supplied by the NeighborhoodManager is callled.

In addition, the **AcceptableMove()** function should be instantiated according to the instantiation of the framework for the vertex cover. The main code of the AcceptableMove() function is following:

```
template<typename Input, typename State, typename Move>
bool MoveRunner<Input, State, Move>:: AcceptableMove()
{
    State tmpState = state;
    neighborhoodManager1→MakeMove(tmpstate, move);

    ...

    for(int i = 1; i < temp1.size();i++)
        if(temp1[i]==1)
        {
            forall_inout_edges(e,vectorOfNodes[i])
                valueOfEdge[index(e)] = 1;
        }

    bool accept = true;
    for(int i = 1; i < valueOfEdge.size(); i++)
        if(valueOfEdge[i] ==0)
            accept = false;

    if(accept == false) return accept;

    if(stateManager1→Objective(state)
        <= stateManager1→Objective(tmpstate);)
        accept = false;
    return accept;
```

}

4.2.5 Solver Classes

The *Solver* class represents the external layer of the framework, controls the search by generating the initial solutions, and deciding how and in which sequence the search has to be activated. In addition, the solvers communicate with the user, by getting the input and delivering the output, and are linked to one or more runner and to some of the managers.

Their code is almost completely provided by framework classes, we have presented the main functions that are supplied by the framework in the subsection 3.3.4. When we use the framework to solve the actual problem, the GenerateNewPopulation() function , UpdatePopulation() function , and other relevant functions should be instantiated according to the instantiation of the framework for *Vertex Cover*.

GenerateNewPopulation() function

The *GenerateNewPopulation* function is the core of *Memetic Algorithms*. Essentially, this function can be seen as a pipeline process comprising several steps. These steps consist of taking several configurations from the previous stage, generating new configurations by applying some operators.

When GenerateNewPopulation function is instantiated for *Vertex Cover*, it includes the following steps:

- **Step 1:** From the previous population, pick out two configurations.
- **Step 2:** Use the 2-Merger (recombination) operator on the two configurations. That is, if one node in the graph is the vertex cover node in both configurations, this node will become one vertex cover node for the new configuration.
- **Step 3:** Use *Greedy Algorithm* 8 in subsection 4.2.1 to pick out some other nodes as the vertex cover node until the new vertex cover is formed for the new configuration.
- **Step 4:** Push the new configuration into the new population.
- **Step 5:** Let one configuration recombines with every other configuration as many times as possible.

The main code of the GenerateNewPopulation function is following:

```
template<typename Input, typename Output, typename State, typename Move>
vector<State> SolverForPopulationSearch <Input, Output, State, Move>
    :: GenerateNewPopulation(vector<State> &pop)
{
    vector<State> tempNewpop;
    int numberOfPop = input.getNumberOfPop();
    int numberOfNode = input.getNumberOfNode();
    State tempState0(numberOfNode);
    vector<int> tempValue0(numberOfNode), tempValue1, tempValue2;

    for(int i = 0; i < numberOfPop-1; i++)
        for(int j = i+1; j < numberOfPop; j++)
        {
            leda_node v;
            leda_edge e;
            vector<leda_node> vectorOfNodes;
            vector<leda_edge> vectorOfEdges;
            leda_graph G = input→getGraph();

            forall_nodes(v,G) vectorOfNodes.push_back(v);
            forall_edges(e,G) vectorOfEdges.push_back(e);

            for(int k=0; k < numberOfNode;k++)
            {
                if(tempValue1[k]==1 && tempValue2[k]==1) tempValue0[k] = 1;
                else tempValue0[k] = 0;
            }
            for(int k=0; k<numberOfNode;k++)
            if(tempValue0[k]==1)
                forall_inout_edges(e,vectorOfNodes[k])
                    G1.del_edge(e);

            int count0=0;
            while(G1.number_of_edges()){
                vector<int> vectorOfEdgeNumber;
                vector<int> vectorOfPosition;

            forall_nodes(v,G1)
            {
                count0=0;
                foral_inout_edges(e,v) count0++;
```

```
                    vectorOfEdgeNumber.push_back(count0);
                }
                count0=vectorOfEdgeNumber[0];
                for(int k = 1; k < vectorOfEdgeNumber.size();k++)
                {
                    if(count0<vectorOfEdgeNumber[k])
                        count0 = vectorOfEdgeNumber[k];
                }
                for(int k = 1; k < vectorOfEdgeNumber.size();k++)
                {
                    if(count0==vectorOfEdgeNumber[k])
                        vectorOfPosition.push_back(k);
                }
                int numberOfCurrentNode=0;
                do{
                    numberOfCurrentNode =
                        vectorOfPosition[int((double(rand())/RAND_MAX)
                        *vectorOfPosition.size())];
                    count0 = 0;
                    forall_inout_edges(e,vectorOfNodes[numberOfCurrentNode])
                        count0++;
                }while(tempValue0[numberOfCurrentNode]!=0 || count0 ==0);

                if(count0 > 0)
                {
                    tempValue0[numberOfCurrentNode] = 1;
                    forall_inout_edges(e,vectorOfNodes[numberOfCurrentNode])
                        G1.del_edge(e);
                }
            }
            for(int k = 0; k < numberOfNode; k++)
                tempState0.setValue(tempValue0[k], k);
            tempNewpop.push_back(tempState0);
        }
        return tempNewpop;
}
```

UpdatePopulation() function

The *UpdatePopulation* function is used to reconstruct the current population using the old population and the new generated population. There are two main possi-

bilities to carry on this reconstruction: the *plus* strategy and the *comma* strategy. In the former, the current population is constructed by taking the best popsize configurations from pop | newpop. In the latter, the best popsize configurations are taken just from newpop,

When the *GenerateNewPopulation* function is instantiated for *Vertex Cover*, it includes the following steps:

- **Step 1:** Receive the old population and the new generated population as the input.
- **Step 2:** Calculate the objective value of the configurations in the old and new population.
- **Step 3:** According to the reconstruction strategy, if the *plus* strategy is selected, concatenate the old population and the new generated population, take the best popsize configurations from it, and refresh the population for the next iteration loop using the best popsize configurations.
- **Step 4:** If the *comma* strategyis selected, just take the best popsize configurations from the new population, and refresh the population for the next iteration loop using the best popsize configurations.

The main code of the UpdatePopulation function is following:

```
template<typename Input, typename Output, typename State, typename Move>
void SolverForPopulationSearch <Input, Output, State, Move>
    :: UpdatePopulation(vector<State> &pop, vector<State> &newpop)
{
    vector<State> tempPop;
    vector<State> tempPop;

    vector<int> objective;
    vector<int> position;

    int i,j,swap,swap1;
    int sizeOfPop = pop.size();
    vint sizeOfNewpop = newpop.size();
    StateManager<Input, State> stateManager0(&input);

    switch(input.getUpdateStrategy())
    {
        case 0:
            for(i = 0; i < sizeOfPop; i++)
```

```
                {
                    objective.push_back(stateManager0.Objective(pop[i]));
                    tempPop0.push_back(pop[i]);
                }
                for(i = 0; i < sizeOfPop; i++)
                {
                    objective.push_back(stateManager0.Objective(pop[i]));
                    tempPop0.push_back(popnew[i]);
                }
                for(i = 0; i < sizeOfPop+sizeOfNewpop; i++)
                    position.push_back(i);
                for(i = 0; i < sizeOfPop; i++)
                    for(j = i+1; j < sizeOfPop+sizeOfNewpop; j++)
                    {
                        swap = objective[i];
                        swap1= position[i];
                        if(objective[i]>objective[j])
                        {
                            objective[i]=objective[j];
                            objective[j]=swap;
                            position[i]=position[j];
                            position[j]=swap1;
                        }
                    }
                for(i = 0; i < sizeOfPop; i++)
                    tempPop.push_back(tempPop0[position[i]]);
                pop = tempPop;
                break;

            case 1:
                ...
                break;
    }
}
```

4.3 The Second Case Study: Independent Set

4.3.1 The Definition and Greedy Algorithm

The maximum **Independent Set** is also a combinatorial optimization problem with many practical applications, such as, the dispersion problems, where we seek a set of mutually separated points. For example, suppose you are trying to identify locations for a new franchise service such that no two locations are close enough to compete with each other. Construct a graph where the vertices are possible locations, and add edges between any two locations deemed close enough to interfere. The maximum *Independent Set* gives you the maximum number of franchises you can sell without cannibalizing sales. *Independent Set* formulated as a decision problem is *NP-Complete*.

Problem 9 *The* **Independent Set** *optimization problem is to find a cover of maximum size in the given graph, i.e., given undirected graph G consisting of nodes V and edges E, G = (V,E), find a maximum size subset of nodes $V_d \in V$ such that no pair of vertices in V_d defines an edge of E between them. V_d is said to be a independent set of Graph G.*

Independent sets avoid conflicts between elements and hence arise often in coding theory and scheduling problems. Define a graph whose vertices represent the set of possible code words, and add edges between any two code words sufficiently similar to be confused due to noise. The maximum independent set of this graph defines the highest capacity code for the given communication channel.

Algorithm 10 (GIS) *The greedy heuristic algorithm for independent set is to randomly select the vertex with lowest degree, add it to the independent set, delete it and all vertices adjacent it, and then repeat until the graph is empty.*

Procedure GIS(Graph G)
begin
    Initialize $V_i \leftarrow \phi$;
    while the graph is not empty
        Randomly select one vertex v with lowest degree;
        Append v to V_i;
        Delete delete it and all vertices adjacent it,
    endwhile
    return V_i
end

Figure 4.2: The Greedy Heuristic Algorithm for Independent Set.

As described in the previous subsection, the core of framework is composed by a set of cooperating classes that include different aspects of the template of *Memetic Algorithms*. These classes in the framework are split in four categories: *Data classes, Manager classes, Runner classes*, and *Solver classes*.

The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, The following part will describe the instantiation of these classes which include all four categories for *Independent Set*. As most encoding procedure for *Independent Set* is similar as one for *Vertex Cover*, only the distinguishable part will be supplied.

4.3.2 Data Classes

The data classes are templates, and therefore they have no actual code. They serve for storing the data, and the data classes need to be instantiated.

In the *Input* class, it includes a graph which can be defined using the facilities of the *LEDA* library, the number of the state in the population, some number of the iteration, and the other relevant data, such as the update Strategy of the population in the memetic algorithm.

In the *Output* class, it includes an integer vector representing the independent set and the other relevant data.

In the *State class*, it includes an integer vector representing the independent set.

In the *Move class*, it includes an integer vector representing the independent set status changing of a node in the graph.

The content of the data classes for Independent Set is similar as that for Vertex Cover.

4.3.3 Manager Classes

The framework provides three manager classes, *StateManager, NeighborhoodManager*, and *OutputManager*, and methods of framework interface classes are in turn split in three categories that we call *MustDef, MayReDef, NoReDef* functions. we need to define MustDef functions, and maybe redefine some MayReDef function as demand.

StateManager Class

In the StateManager class in the application for *Independent Set*, **RandomState** function and **Objective** function need to be defined.

RandomState function is for generating a initial state randomly. It is $MustDef$ function to be instantiating.

For the problem of Independent Set, the *Greedy Algorithm* 10 in subsection 4.3.1 is used for generating individual initial state.

4.3.4 Runner Classes

The **Runner** classes are responsible for performing a run of a *local search* technique, starting from one initial individual state in the population and leading to the final one.

Each runner has many data objects for representing the state of the search, such as current state, current move, the number of iterations, etc., and maintains links to all the managers, which are invoked for performing specific tasks on its own data. The example of runners is hill climbing search.

The framework comprises a hierarchy of runners. The base class Runner has only Input and State templates, and it is connected to the solvers, which have no knowledge about the neighborhood relations.

The class *MoveRunner* requires also the template *Move*, and the pointers to the necessary managers. It also stores the basic data common to all derived classes: the current state, the current move, and the number of iterations.

In the subsection 3.3.3, we present the main code of *Go()*, the main function of MoveRunner, which performs a full run of *local search*.

Most of the functions invoked by *Go()* are defined in the subclasses of *MoveRunner*. It depends on what kind of search strategy will be used in the local search, what kind of *NeighborhoodManager* will be pointed to.

4.3.5 Solver Classes

Solvers represent the external layer of the framework, control the search by generating the initial solutions, and deciding how and in which sequence the search has to be activated. In addition, the solvers communicate with the user, by getting the input and delivering the output, and are linked to one or more runner and to some of the managers.

Their code is almost completely provided by framework classes, we have presented the main functions that are supplied by the framework in the subsection

3.3.4. When we use the framework to solve the actual problem, the *GenerateNewPopulation*() function , *UpdatePopulation*() function ,and other relevant functions should be instantiated according to the instantiation of the framework for *Independent Set*.

GenerateNewPopulation() function

The *GenerateNewPopulation* function is the core of the memetic algorithm. When *GenerateNewPopulation* function is instantiated for *Independent Set*, it includes the following steps:

- **Step 1:** From the previous population, pick out two configurations.
- **Step 2:** Use the 2-Merger (recombination) operator on the two configurations. That is, if one node in the graph is the independent set node in both configurations of the previous population, then this node will become one independent set node for the new configuration.
- **Step 3:** Use *Greedy Algorithm* 10 in subsection 4.3.1 to pick out some other nodes as the independent set node until the new independent set is formed for the new configuration.
- **Step 4:** Push the new configuration into the new population.
- **Step 5:** Let one configuration recombines with every other configuration as many as possible.

UpdatePopulation() function

The *UpdatePopulation* function is used to reconstruct the current population using the old population and the new generated population. There are two main possibilities to carry on this reconstruction: the *plus* strategy and the *comma* strategy. In the former, the current population is constructed by taking the best popsize configurations from pop | newpop. In the latter, the best popsize configurations are taken just from newpop,

When UpdatePopulation() function is instantiated for *Independent Set*, it includes the following steps:

- **Step 1:** Receive using the old population and the new generated population as the input.
- **Step 2:** Calculate the objective value of the configurations in the old and new population.

- **Step 3:** According to the reconstruction strategy, if the *plus* strategy selected, concatenate the old population and the new generated population, take the best popsize configurations from it, and refresh the population for the next iteration loop using the best popsize configurations.
- **Step 4:** If the *comma* strategy selected, take just the best popsize configurations from the new population, and refresh the population for the next iteration loop using the best popsize configurations.

Chapter 5

Testing and Experimental Results

In Chapter 2, a general template for *Memetic Algorithm* was presented and we believe that the use of object-oriented framework can be helpful in designing and implementing this template for some combinational optimization problem.

In Chapter 3, an object-oriented framework for *Memetic Algorithm* was developed and the more detailed structures and main class components of the framework were supplied. A framework is a special kind of software library, which consists of a hierarchy of abstract classes. When the user wants to use the framework developing one's own application for solving one special NP-Hard optimization problem, he only needs to define suitable derived classes, which implement the virtual functions of the abstract classes, and supply the problem specific details.

In the last chapter, we presented in detail how to encode the framework of Memetic Algorithms to form applications for two well-known combinatorial optimization problems, *Vertex Cover* and *Independent Set*, and supplied some detailed codes in *C++*.

In this chapter, we will do testing on some graphs using the applications developed for *Vertex Cover* and *Independent Set*, and supply some testing results. These graphs include some random graphs, some benchmark graphs, and Papadimitriou and Steiglitz's Regular Graph special for *Vertex Cover*.

5.1 The First Case Study: Vertex Cover

A general template for *Memetic Algorithm* is listed below:

```
Procedure Population-Based-Search-Engine
begin
    Initialize pop using GenerateInitialPopulation();
```

```
    repeat
        newpop ← GenerateNewPopulation(pop);
        pop ← UpdatePopulation(pop,newpop);
        if pop has converged then
            pop ←RestartPopulation(pop);
        endif
    until TerminationCriterion();
end
```

Figure 5.1: A Population-Based Search Algorithm.

According to the above template, the whole rough search procedure of population-based search algorithm should include the following steps:

- **Step 1** According to the *Fig.* 2.3 "Creating high-quality solutions in the initial population", first of all, the GenerateInitialPopulation procedure creates the initial set of |pop| configurations. This is done by using the *Greedy Algorithm* 8 for Vertex Cover to generate one configuration or possibly, the Local-Search-Engine presented in subsection 2.2.3 could be used as shown in *Fig.* 2.1, by means of which high-quality configurations are created in the initial population.
- **Step 2** Based on the previous generation population, the GenerateNew-Population function generates some new configurations, the detailed step of instantiating it for Vertex Cover is listed in subsection 4.2.5.
- **Step 3** Based on the old population and the new generated population that were generated by GenerateNewPopulation function, the UpdatePopulation function reconstructs the current population. There are two main possibilities to carry on this reconstruction: the *plus strategy* and *the comma strategy.* The detailed step of instantiating it for Vertex Cover is listed in subsection 4.2.5.
- **Step 4** When some termination criteria are satisfied, such as setting a limit on the total number of iterations, reaching a maximum number of iterations without improvement, or having performed a certain times of population restarts, etc, the whole search procedure will stop, or go back to **Step 2**.

5.1.1 Testing on Some Randomly Generated Graphs

Random Graph One

For the purpose of evaluating the application instantiated from the framework, a random graph , the number of whose *nodes* and *edges* is **1500** and **4500** respec-

tively, is generated.

Here, the number of configurations in population is set **15**.

After using *Greedy Algorithm* 8, the initial vertex cover configurations are produced, and the *objective function* set of individual configurations is {**934, 932, 936, 944, 936, 933, 933, 940, 937, 940, 938, 933, 936, 941, 942**}.

After using the *local search algorithm*(such as hill climbing), there is nearly no improvement to the individual configurations. The main reason for the non-improvement is that for the vertex cover problem the result of the initial configuration using the greedy algorithm is good enough, and normal local search algorithm is not strong enough to improve them.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **902**.

Random Graph Two

Another random graph, the number of whose nodes and edges is **1500** and **3000** respectively, is generated. When the number of configurations is set in different values, the final solution of objective values of vertex cover is different. Increasing the number of configurations sometimes may improve the approximation.

1.When the number of configurations is set **15**

After using *Greedy Algorithm* 8, the initial vertex cover configurations are produced, and the objective set of individual configurations is {**818, 819, 820, 814, 819, 821, 815, 824, 813, 820, 812, 822, 817, 817, 822**}.

After using the *local search algorithm*, there is nearly no improvement to the individual configurations.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **792**.

2. When the number of configurations is set **30**

After using *Greedy Algorithm* 8, the initial vertex cover configurations are produced, and the objective set of individual configurations is {**818, 819, 820, 814, 819, 821, 815, 824, 813, 820, 812, 822, 817, 817, 822, 819, 816, 819, 817, 809, 817, 815, 813, 819, 816, 816, 815, 816, 817, 818**}.

After using the *local search algorithm*, also there is nearly no improvement to the individual configurations.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **790**.

5.1.2 Testing on Benchmark Graphs

In a paper "Evolutionary Algorithm for Vertex Cover" [E98], Isaac introduces some stochastic optimization approaches and supplies some experimental results on some benchmark graphs.

The work reported in that paper makes use of an alternative traditional heuristic for Vertex Cover, labeled **OBIG**. Rather than choosing the vertex of maximum remaining degree to include in a cover, the inverse greedy approach to vertex cover chooses the vertex of minimum remaining degree to exclude from cover. Each excluded vertex forces adjacent vertices into the cover to maintain feasibility of the solution.

In the encoding of the underlying problem for Evolutionary Algorithm, the most obvious approach is a direct encoding in which each bit of a binary chromosome of length |V| defines the presence or absence of the corresponding vertex in the cover. One approach labeled **BDD-IG** taken in that paper uses a fitness evaluation skeleton based on the above OBIG heuristic with embedded binary decision diagram (BDD) encoding. Each bit of the chromosome corresponds not to a particular vertex, but rather to the next decision to be made during synthesis of a feasible cover.

Another traditional elitist Genetic Algorithm was implemented using the **BDD-IG** encoding/fitness evaluation function. The resulting Genetic Algorithm is labeled **BDDGA**.

Both OBIG and BDD-IG encoded evolutionary algorithm have been empirically tested using the VC benchmarks described in [PK95]. Pramanick investigated two graph classes using **PDI**: a class of random graphs (irand##) and a class of graphs (sg##) and compared PDI algorithm performance against traditional heuristics for vertex cover, including greedy (**GH**) and maximal matching (**MM**).

In my thesis, we also perform testing using the application developed for Vertex Cover on the above two graph classes of the VC benchmark. The testing results on the benchmark graph are summarized in *Table* 5.1 in page 55.

5.1.3 Testing on Papadimitriou and Steiglitz's Regular Graphs

The regular graph after Papadimitriou and Steiglitz consists of three levels; the first two have the same number of nodes while the third level has two nodes less than the previous two levels. More precisely, each graph consists of $n = 3k + 4$ (k≥1) nodes, $k + 2$ nodes on the first level, labeled by *1, ... , k + 2*; followed by $k + 2$ nodes on the second level, labeled $k + 3$, *... , 2k + 4*, while the k nodes of the third level are labeled *2k + 5, ... , 3k +4*.

Table 5.1: Testing results for VC on two graph classes of the VC benchmark.

Graph	GH	PDI	OBIG	BDDGA	MEMETIC
irand 500	402	369	353		353
irand 1000a	688	627	591	588	590
irand 1000b	795	723	698	694	697
irand 1300	1110	1045	1014	1009	1010
sg 269	190	177	173	172	173
sg 698	494	465	450	446	415
sg 821	465	433	392	391	397
sg 1742	1206	1124	1108	1100	1105
sg 1770	1114	1049	1063	1047	1013
sg 2083	1246	1157	1149	1139	1143

The regular graph for $k = 3$ is produced in *Fig.* 5.2.

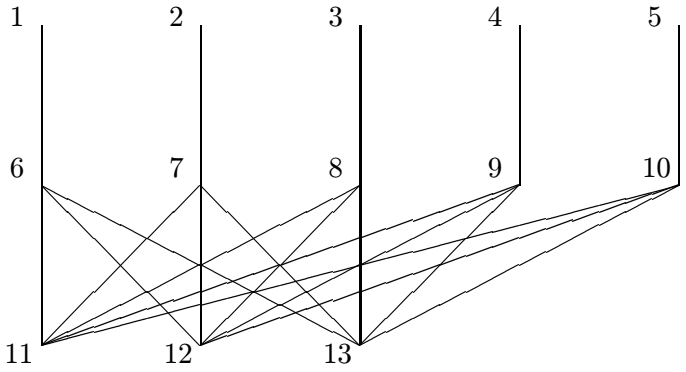

Figure 5.2: Construction of the Regular Graph after Papadimitriou and Steiglitz.

Fact 11 *The minimum vertex cover the regular graph after Papadimitriou and Steiglitz should include all nodes of the second level.*

Produce a regular graph for $k = 3$, then:
The degree for the nodes in the first level is **1**
The degree for the nodes in the second level is k + 1 = **4**
The degree for the nodes in the third level is k + 2 = **5**

As described in the subsection 5.1, the first step of the framework application for Vertex Cover is to get the initial configurations of the population by the *Greedy Algorithm* 8 which would start with the nodes of the third level since these nodes have the highest degree. Consequently, the greedy strategy finds the solution of vertex cover as **8**.

Here, the number of initial configurations of the population is set to *15*.

The running results of initial configurations are summarized in *Table* 5.2 in page 56:

Table 5.2: The initial configurations by using Greedy Algorithms.

No. of Init Configurations	**Vertex Cover**	**Objective**
1	0101010101111	8
2	0001011101111	8
3	1011001001111	8
4	0110010011111	8
5	0010111010111	8
6	0010111010111	8
7	1011001001111	8
8	1010001011111	8
9	0000111110111	8
10	0001111100111	8
11	1100000111111	8
12	0001011101111	8
13	0101010101111	8
14	0110110010111	8
15	0110110010111	8

The common characteristic of vertex covers list in *Table* 5.2 is that they all include the last three bits that represent the third level of the regular graph.

Then the population search algorithm that uses Merger (recombination) operator is used at the following step. But the objective functions of the individual configurations have no improvement no matter how many generations are included.

The following is the population search result:

The running results after population search are summarized in *Table* 5.3 on page 57:

The main reason for the above results is that Merger operator regards one node as a vertex cover node for the new configuration, if that node is the vertex cover

Table 5.3: The running results of configurations after population search.

No. of Configurations	Vertex Cover	Objective
1	0101010101111	8
2	0001011101111	8
3	1011001001111	8
4	0110010011111	8
5	0010111010111	8
6	0010111010111	8
7	0010111010111	8
8	1011001001111	8
9	1010001011111	8
10	0000111110111	8
11	0001111100111	8
12	1100000111111	8
13	0101010101111	8
14	0110110010111	8
15	0110110010111	8

node in all parent configurations, and use the greedy algorithm again to pick out some other nodes as the vertex cover node until the new vertex cover is formed for the new configuration.

The solution to solve the problem described above is to insert the Local Search strategy after every greedy algorithm on the initial individual configurations.

The result of the initial configuration of the population after using the Local Search strategy are summarized in *Table* 5.4 on page 58:

Finally, the population search algorithm is used based on the above initial population listed in *Table* 5.4. All final configurations in population will converge to the same result:

The vertex cover = {**0000011111000**}, the objective value = **5**.

The final converged result shows that the vertex cover only includes the second level of the regular graph.

5.2 The Second Case Study: Independent Set

According to the template listed in section 5.1, the whole rough search procedure for *Independent Set* includes the following steps:

Table 5.4: The initial configurations after using Local Search.

No. of Init Configurations	Vertex Cover	Objective
1	0101010101111	8
2	0001011111001	7
3	0000111111001	7
4	1101000101111	8
5	0111110000111	8
6	1000011111001	7
7	0001111111000	7
8	0001111100111	8
9	0001111100111	8
10	0000011111100	6
11	0011011001111	8
12	0000111111000	6
13	0000011111110	8
14	0011011001111	8
15	1010001011111	8

- **Step 1** According to the *Fig.* 2.3 "Creating high-quality solutions in the initial population", first of all, the GenerateInitialPopulation procedure creates the initial set of |pop| configurations. This is done by using the *Greedy Algorithm* 10 for *Independent Set* to generate one configuration or possibly, the Local-Search-Engine presented in subsection 2.2.3 could be used as shown in *Fig* 2.1, by means of which high-quality configurations are created in the initial population.

- **Step 2** Based on the previous generation population, the GenerateNewPopulation function generates some new configurations, the detailed step of instantiating it for *Independent Set* is listed in subsection 4.2.5.

- **Step 3** Based on the old population and the new generated population that were generated by GenerateNewPopulation function, the UpdatePopulation function reconstructs the current population. There are two main possibilities to carry on this reconstruction: the *plus strategy* and the *comma strategy.*

- **Step 4** When some termination criteria are satisfied, such as setting a limit on the total number of iterations, reaching a maximum number of iterations without improvement, or having performed a certain number of population restarts, etc, the whole search procedure will stop, or go back to **Step 2**.

5.2.1 Testing on Some Randomly Generated Graphs

Random Graph One

A graph the number of whose *nodes* and *edges* is **1500** and **4500** respectively is generated randomly. (This random graph is same as the random one used in subsection 5.1.1 for vertex cover)

Here, the number of the configuration is set **15**.

After using *Greedy Algorithm* 10, the initial independent set configurations are produced, and the objective function set of individual configurations is {**592, 591, 593, 591, 586, 592, 594, 593, 589, 596, 591, 591, 586, 590, 590**}.

After using the *local search* algorithm (such hill climbing), there is nearly no improvement to the individual configurations. The main reason for the non-improvement is that for the independent set problem the result of the initial configuration using the greedy algorithm is good enough, and normal local search algorithm is not strong enough to improver them.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **608**.

Random Graph Two

Another graph, the number of whose *nodes* and *edges* is **1500** and **3000** respectively, is generated randomly. (This random graph is the same as the random one used in subsection 5.1.1 for vertex cover)

1. When the number of the configuration is set **15**.

After using *Greedy Algorithm* 10, the initial independent set configuration is produced, and the objective function set of individual configurations is {**703, 713, 705, 709, 709, 709, 706, 704, 709, 710, 710, 712, 701, 710, 705**}.

After using *local search* algorithm, also there is nearly no improvement to the individual configurations.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **721**.

2. When the number of the configuration is set **30**.

After using *Greedy Algorithm* 10, the initial independent set configuration is produced, and the objective function set of the individual configuration is {**703, 713, 705, 709, 709, 709, 706, 704, 709, 710, 710, 712, 701, 710, 705, 703, 710, 698, 712, 708, 708, 708, 712, 712, 713, 711, 710, 708, 708, 706**}.

After using the *local search* algorithm, also there is nearly no improvement to the individual configurations.

After using the Merger (recombination) operator, the final configurations in the population are converged to the same objective value **721**.

5.2.2 Testing on Benchmark Graphs

In a paper "Evolutionary Algorithm for Vertex Cover" [E98], Isaac supplies some experimental results on the *DIMACS* maximum clique benchmark graphs. Two of the DIMACS graph classes are tested here: *keller* and *mann*.

A hybrid algorithm which incorporates a mutation-only hill climbing post-processing phase (*HC*) with the recombination-only Hypergamous Parallel GA (*HPGA*) is labeled as **HPGA+HC** in that paper.

Some testing performances of the *OBIG* algorithm and a *BDD-ID* encoded *HPGA+HC* on two *DIMAC* graph classes are shown in that paper.

Also, some testing results for Aggarwal's OCH GA for *Independent Set* and Bui's GA for the maximum clique [BE95] are reported. There are two versions of **OCH**: **OCH2** corresponds to the best solution of two experiments; **OCH20** reports the best solution of twenty experiments.

In my thesis, we also perform testing using the application developed for *Independent Set*. Each result for *Memetic* listed in *Table* 5.5 was run on the complementary graphs of the original maximum clique benchmark.

The testing results on DIMACS graph classes are summarized in *Table* 5.5 on page 60.

Table 5.5: Testing results for Independent Set on DIMACS graphs

Graph	OCH2	OCH20	GMAC	OBIG	HPGA +HC	MEMETIC
mann_a9	16			16		16
mann_a27	126	126	125	126		126
keller4	11	11	11	11		11
keller5	25	27	18	26	26	27

Chapter 6

Conclusion

We now summarize our thesis as follows.

- There are a lot of such problems in industry and science, many optimization problems are fundamentally *hard*. One of the most accepted ways to prove that a problem is hard is to prove it NP-complete. If an optimization problem is NP-complete we are almost certain that it cannot be solved optimally in polynomial time. In practice, however, the quest to solve hard problems is not quite so hopeless as this definition suggests. This is due to the use of *approximate methods*.

  An approximate method is an algorithm that we use to try to find solutions to hard optimization problem, and which runs quickly, but which gives no guarantee that the solution it will find is the best one. Good approximate methods, especially when tailored and customized to work with a particular optimization problem, may often find a good solution very quickly.

  The existing, successful methods in approximate iterative optimization fall into two board classes: *local search*, and *population-based search*. In *local search*, a special 'current' solution is maintained, and its neighbors are explored to find better quality solutions. Occasionally, one of these neighbors becomes the new current solution, and then its neighbors are explored, and so forth. In *population-based search*, the notion of a single current solution is replaced by the maintenance of a population of different current solutions. New solutions are generated by first selecting members if this population to be parent, and then making changes to these parents to produce children. In population-based search, since there is now a collection of current solutions, rather than just one, we can exploit by using two or more from this collection at once as the basis for generating new candidates.

- There are many *population-based* optimization algorithms and variant ways to handle the above issues. In this thesis, a special emphasis has been focused

on *Memetic Algorithms*, which represents one of the more successful emerging ideas in the ongoing research effort to understand population based and local search algorithms.

Memetic Algorithms are evolutionary algorithms that apply a separate local search process to refine individuals. Memetic Algorithms includes a broad class of *metaheuristics*. This method is based on a population of agents and proved to be of practical success in a variety of problem domains. We can be sure that Memetic Algorithms constitute one of the most successful approaches for combinatorial optimization in general, and for the approximate solution of *NP* Optimization problems in particular.

From an optimization point of view, Memetic Algorithms are hybrid evolutionary algorithms that combine global and local search by using an evolutionary algorithm to perform exploration while the local search method performs exploitation. Combining global and local search is a strategy used by many successful global optimization approaches,

According to the above considerations, a general template for a memetic algorithm was provide. This is the core algorithm of this thesis.

- Based on a general template for *Memetic Algorithms* supplied, A object-oriented $framework$ was developed in $C++$ to help in designing and implementing this template. A framework is a special kind of software library, which consists of a hierarchy of abstract classes. The user needs only to define suitable derived classes, which implement the virtual function of the abstract classes.

  The basic idea for the development of object-oriented application is the exploitation of already available Design patterns. They are abstract structures of classes that are commonly presented in object-oriented applications and frameworks that have been precisely identified and classified. Their use allows the designer to address many implementation issues in a more principled way.

  In this thesis, an object-oriented framework was present to be used as a general tool for the development and the implementation of memetic algorithm in $C++$. The basic idea of framework is to capture the essential features of most memetic algorithm techniques, and their possible compositions.

- As examples of actual use of the framework, we present the development of the framework of Memetic Algorithms for two well-known combinatorial optimization problem, one is *Vertex Cover*, the other is *Independent Set*.

  The core of the framework is composed of a set of cooperating classes that take care of different aspects of the template of Memetic Algorithms. These classes in the framework are split in four categories: *Data classes, Manager*

classes, Runner classes, and *Solver classes*, and methods of framework interface classes are in turn split into three categories that we call *MustDef, MayReDef, NoReDef* functions. we need to define $MustDef$ functions, and maybe redefine some $MayReDef$ function as demand.

The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithm.

Bibliography

[ACR97] A.A. Andreatta, S.E.R. Carvalho, and C.C. Ribeiro. A framework for the development of local search heuristics for combinatorial optimization problems. In *Proc. of the 2nd Metaheuristics International Conference,* 1997.

[AOT97] C.C. Aggarwal, J.B. Orlin, and R.P. Tai, . Optimized Crossover for the Independent Set Problem. *Oerations Research*,45(2) 226-234. March-April 1997.

[BE95] T.N. Bui and P.H. Eppley. *A Hybrid Genetic Algorithm for the Maximum Clique Problem.* in Proceedings of the 6th ICGA, Pages 478-484, Morgan Kaufmann. 1995.

[BFS99] B. De Backer, V. Furnon, and P. Shaw. An object model for meta-heuristic search in con-straint programming. In *Workshop On Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, (CP-AI-OR99), 1999.

[BKDG96] A. de Bruin, G. Kindervater, H. Trienekens, R. van der Goot, and W. van Ginkel. An obiect oriented approach to generic branch and bound. Technical Report EUR-FEW-CS-96-10, Erasmus University, Department of Computer Science, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands, 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacoson. *The unified modeling language user guide.* Addison Wesley, Reading(mass), 1999.

[CD94] Kevin Cattell and Michael J. Dinneen. A characterization of graphs with vertex cover up to five. In Vincent Bouchitte and Michel Morvan, editors, *Orders, Algorithms and Applications, ORDAL'94*, volume 831 of *Lecture Notes on Computer Science*, pages 86–99. Springer-Verlag, July 1994.

[CDG99] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization.* McGraw-Hill. 1999.

[CG93] H.G. Cobb and J.J. Grefenstette. Genetic algorithms for tracking changing environments. In *Proceedings of Workshop on Graph-Theoretic Con-*

cepts in Computer Science WG'99, volume 1665 of *Lecture Notes on Computer Science*, pages 313–324. Springer-Verlag, 1999.

[CKJ99] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. In S. Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, pages 529–530, San Mateo, CA, 1993. Morgan Kaufmann.

[CL86] Gary Chartrand and Linda Lesniak. *Graphs and Digraphs*. Wadsworth Inc., 2nd edition, 1986.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L.Rivert. *Introduction to Algorithm*. The MIT Press, Cambridge, MA. 1990.

[CTG97] T.G. Crainic, M. Toulouse, and M. Gendreau. Toward a taxonomy of parallel tabu search heuristics. *INFORMS Journal of Computing*, 9(1):61 72, 1997.

[Daw76] R. Dawins. *The Selfish Gene*. Clarendon Press, Oxford, 1976.

[DBK92] Y. Davidor and O. Ben-Kiki. The interplay among the genetic algorithm operators: Information theory tools used in a holistic way. In R. Manner and B. Manderick, editors,*Parallel Problem Solving From Nature* II, pages 75-84. Amsterdam, 1992. Elsevier Science Publishers B.V.

[Din99] M.J. Dinneen An Introduction to Graph Algorithms. Department of Computer Science, The University of Auckland. 1999.

[E98] Isaac K. Evans. Evolutionary Algorithms for Vertex Cover. In *Evolutionary Programming VII,Proceedings Severnth International Conference*, 1999.

[ECS89] L.J. Eshelman, R.A. Caruna, and J.D. Schaffer. Biases in the crossover landscape. In J.D.Schaffer, editors,*Proceedings of the 3rd International Conference on Genetic Algorithm.*, pages 10-19. San Mateo, CA, 1989. Morgan Kanfmann Publishers.

[FHL96] J. A. Ferland, A. Hertz, and A. Lavoie. An object-oriented methodology for solving assignment type problems with neighborhood search techniques. Operations Research,44(2):347 359, 1996.

[FVW98] A. Fink, S. Vo, and D.L. Woodruff. Building reusable software components for heuristc search, 1998. Extended abstract of the talk given at OR98, Zurich.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattern, Elements of Resuable Object-Oriented Software*. Addison Wesley, Reading(mass), 1994.

[GS00] L.D. Gaspero and A. Schaerf. *EASYlOCAL++: An Object-oriented framework for flexible design of local search algorithms*. Research Report 2000.

[HM98] P. Hansen and N. Mladenovic. An introduction to variable neighbourhood search. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-Heuristics Advances and Trends in Local Search Paradigms for Optimization*, pages 433-458. Kluwer Academic Publishers, 1999.

[LYY99] S.J. Louis, X. Yin, and Z.Y. Yuan. Mutilple vehicle routing with time windows using genetic algorithms. In *Preceedings of the 1999 Congress on Evolutionary Computation*, page 1804-1808, Washington D.C., 1999. IEEE Neural Network Council - Evolutionary Programming Society - Institution of Electrical Engineers.

[MNU99] K. Michel, S. Naher, and M. Uhrig. *The LEDA User Manual*, 1999 Version.

[Mos01] Pablo Moscato. *NP Optimization Problems, Approximability and Evolutionary Computation: From Practice to Theory.* in Ph.D. Dissertation, Universidade Estadual de Campinas, Brazil. March 2001.

[PK95] I. Pramanick and J.G. Kuhl. An inherently Parallel Method for Heuristic Problem-Solving: Part II Example Application. *IEEE Transactions on Parallel and Distributed System, 6(10):1016-1028,* 1995.

[SR96] P.D. Surry and N.J. Radcliffe. Inoculation to initialise evolutionary search. In T.C. Fogarty, editor, *Evolutionary Computing: AISB Workshop*, number 1143 in Locture Notes in Computer Science, pages 269-285. Springer-Verlag, 1996.

[SSP95] A. Schappert, P. Sommerland, and W. Pree. Automated framework development. In *Symposium on Software Reusability (ACM Software Engineering Notes)*, August 1995.

[VMOR99] S. Voss, S. Martello, I. Osman, and C. Roucairol, editors. Meta-Heuristics Advances and Trends in Local Search Paradigms for Optimization. Kluwer Academic Publishers, 1999.