EASY LOCAL++:
an object-oriented framework
for the flexible design of
local-search algorithms

Luca Di Gaspero\(^1\), *\(^∗\), † and Andrea Schaerf\(^2\)

\(^1\)Dipartimento di Matematica e Informatica, Università degli Studi di Udine, Italy
\(^2\)Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica, Università degli Studi di Udine, Italy

**SUMMARY**
Local search is a paradigm for search and optimization problems, which has recently evidenced to be very effective for a large number of combinatorial problems. Despite the increasing interest of the research community in this subject, there is still a lack of a widely-accepted software tools for local search.

We propose EASY LOCAL++, an object-oriented framework for the design and the analysis of local-search algorithms. The abstract classes that compose the framework specify and implement the invariant part of the algorithm and are meant to be specialized by concrete classes that supply the problem-dependent part. The framework provides the full control structures of the algorithms, and the user has only to write the problem-specific code. Furthermore, the framework comes with some tools that simplify the analysis of the algorithms.

The architecture of EASY LOCAL++ provides a principled modularization for the solution of combinatorial problems by local search and helps the user by deriving a neat conceptual scheme of the application. It also supports the design of combinations of basic techniques and/or neighborhood structures.

The framework has been tested in some applicative domains and has proved to be flexible enough in the implementation of algorithms for the solution of various scheduling problems. Copyright © 2003 John Wiley & Sons, Ltd.

**KEY WORDS:** algorithms design and implementation; (meta-)heuristics; local search; analysis of algorithms

\(^∗\)Correspondence to: Luca Di Gaspero, Dipartimento di Matematica e Informatica, Università degli Studi di Udine, via della Scienze 206, I-33100 Udine, Italy.
\(^†\)E-mail: dgasper@dimi.uniud.it

Contract/grant sponsor: Optiware Italia
Contract/grant sponsor: Università degli Studi di Udine, Progetto Giovani Ricercatori 2000

Published online 15 May 2003
Copyright © 2003 John Wiley & Sons, Ltd.
INTRODUCTION

Local search is a family of methods used to find approximate solutions for hard combinatorial optimization problems. This paradigm is based on the idea of navigating the search space by iteratively stepping from one state to one of its ‘neighbors’, which are obtained by applying a simple local change to the current solution.

Differently from other search paradigms, e.g. branch and bound, no widely-accepted software tool is available up to now for local search; only a few research-level prototypes have gained limited popularity. In our opinion, the reason for this lack is twofold. On the one hand, the apparent simplicity of local search induces the users to build their applications from scratch. On the other hand, the rapid evolution of local-search techniques (tabu search, variable neighborhood search, iterative local search, etc.) seems to make the development of general tools impractical.

We believe that the use of object-oriented (OO) frameworks can help in overcoming this problem. A framework is a special kind of software library, which consists of a hierarchy of abstract classes. The user only defines suitable derived classes that implement the virtual functions of the abstract classes. Frameworks are characterized by using the inverse control mechanism (also known as the Hollywood principle: ‘Don’t call us, we’ll call you’) for the communication with the user code: the functions of the framework call the user-defined functions and not the other way round. The framework thus provides full control structures for the invariant part of the algorithms and the user only supplies the problem-specific details. In contrast, libraries that use a direct control mechanism, such as LEDA [1], are called toolkits in the OO jargon.

In this paper we present EASYLOCAL++, an OO framework that can be used as a general tool for the development and the analysis of local-search algorithms in C++. The basic idea behind EASYLOCAL++ is to capture the essential features of most local-search techniques and their possible compositions.

EASYLOCAL++ relies on two design patterns [2]. These are abstract structures of classes, commonly present in OO applications and frameworks, that are commonly present in OO 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. Namely, EASYLOCAL++ is based on the template method, which specifies and implements the invariant parts of various search algorithms, and the strategy method, which communicates between the main solver and its component classes.

The framework provides a principled modularization for the design of local-search algorithms and exhibits several advantages with respect to directly implementing the algorithm from scratch, not only in terms of code reuse but also in methodology and conceptual clarity. Moreover, EASYLOCAL++ is fully glass-box and is easily extensible by means of new class derivations and compositions. The above features mitigate some potential drawbacks of the framework, such as the computational overhead and the loss of full control in the implementation of the algorithms.

EASYLOCAL++ is part of the LOCAL++ project, which aims at realizing a set of OO software tools for local search. Further information about the project is available on the Web at http://www.diegm.uniud.it/schaerf/projects/local++. From the same address it is possible to freely download a stable and documented version of EASYLOCAL++.

The use of EASYLOCAL++ in the solution of a classical problem, specifically the GRAPHCOLORING problem, is exemplified in our case study. In addition, the use of the framework...
in the development of a real application (namely, a solver for the University Course Timetabling problem) is described thoroughly in a companion work [3] that has recently appeared in [4].

**LOCAL SEARCH**

Local search is a family of general-purpose techniques for search and optimization problems. These techniques are *non-exhaustive* in the sense that they do not guarantee that a feasible (or optimal) solution will be found, but they explore a search space until a specific stop criterion is satisfied.

Local-search methods are usually considered as *meta-heuristics*, since they describe a general technique for driving a underlying heuristic (dependent on the problem at hand) towards a good solution.

**Local-search basics**

Given an instance $p$ of a problem $P$, we associate a *search space* $S$ with it. Each element $s \in S$ corresponds to a potential solution of $p$ and is called a *state* of $p$. Local search relies on a function $N$ (depending on the structure of $P$), which assigns to each $s \in S$ its *neighborhood* $N(s) \subseteq S$. Each state $s' \in N(s)$ is called a *neighbor* of $s$.

A local-search algorithm starts from an initial state $s_0$ (which can be obtained with some other technique or generated randomly) and enters a loop that *navigates* the search space, stepping from a state $s_i$ to one of its neighbors $s_{i+1}$.

Several search control strategies can be defined on this framework, according to the criteria used for the selection of the move and for stopping the search. However, in all techniques the search is driven by a *cost function* $f$ that estimates the quality of each state and, without loss of generality, has to be minimized. For constraint satisfaction problems, $f$ generally accounts for the number of violated constraints, whereas for optimization problems it also takes into account the objective function of the problem.

The most common local-search techniques are *hill climbing*, *simulated annealing* and *tabu search*. These techniques have many variants and we briefly describe here the version that is implemented within the basic classes of EASY LOCAL+++. The reader who is interested in a more detailed presentation can refer, for example, to [5] and [6].

**Hill climbing**

Hill climbing is actually not a single local-search technique, but rather a family of techniques based on the idea of only performing moves that improve (or leave unchanged, i.e. *sideways* moves) the value of the cost function $f$. More formally, a hill-climbing strategy selects a move $m_i$ at each iteration $i$ and if $f(s_i \oplus m_i) < f(s_i)$ (or $f(s_i \oplus m_i) \leq f(s_i)$), then let $s_{i+1} = s_i \oplus m_i$, otherwise let $s_{i+1} = s_i$.

The way a move is selected and whether or not sideways moves are accepted characterize the different hill-climbing strategies. For example, the so-called *steepest descent* strategy relies on the
exhaustive exploration of the neighborhood and stops as soon as no improving move is available, i.e. a local minimum has been reached.

In general, not all hill-climbing strategies stop when they reach a local minimum. In fact, whenever hill climbing also accepts sideways moves, the search might loop infinitely by cycling between two or more local minima neighbors at equal costs.

To guard against this situation, the stop criterion is based on the number of iterations elapsed from the last strict improvement. Specifically, given a fixed value \( n \) the algorithm stops after \( n \) iterations that do not improve the value of the cost function, i.e. it stops at iteration \( j \) such that \( f(s_j) = f(s_{j-1}) = \cdots = f(s_{j-n}) \).

Simulated annealing

Simulated annealing is a probabilistic local-search technique whose name comes from the fact that it simulates the cooling of a collection of hot vibrating atoms.

The process starts by creating a random initial state \( s_0 \). At each iteration \( i \) the candidate move \( m_i \) from \( s_i \) is generated at random.

Letting \( \Delta_i \) be the difference \( f(s_i \oplus m_i) - f(s_i) \), if \( \Delta_i < 0 \) the new solution is accepted and becomes the current state (i.e. \( s_{i+1} = s_i \oplus m_i \)). If \( \Delta_i \geq 0 \) (that is, \( m_i \) is a non-improving move), the new solution is accepted with probability \( e^{-\Delta_i/T} \), where \( T > 0 \) is a parameter, called the temperature. Conversely, with probability \( 1 - e^{-\Delta_i/T} \) the move is not accepted and the current state remains unchanged, i.e. \( s_{i+1} = s_i \).

At the beginning of the process, the temperature \( T \) is set to a given high value \( T_0 > 0 \). Then, after a fixed number of iterations, the temperature decreases by a cooling rate \( \alpha \) (0 < \( \alpha < 1 \)), so that \( T_n = \alpha T_{n-1} \).

The entire search stops when the temperature reaches a value very close to zero, hence no solution that does not improve the cost function could be accepted. In this case we say that the system is frozen and the solution obtained at that temperature is obviously a local minimum.

The control knobs of the procedure are the cooling rate \( \alpha \), the number of states sampled at each temperature and the starting temperature \( T_0 \).

Tabu search

Tabu search is a local-search strategy based on memory. At each state \( s_i \), tabu search explores a subset \( V \) of the current neighborhood \( N(s_i) \). Among the elements in \( V \), the one that gives the minimum value of the cost function becomes the new current state \( s_{i+1} \), independent of the fact of whether \( f(s_{i+1}) \) is less or greater than \( f(s_i) \).

Such a choice allows the algorithm to escape from local minima, but creates the risk of cycling among a set of states. In order to prevent cycling, the so-called tabu list is used, which determines the forbidden moves. This list stores the most recently accepted moves and the inverses of the moves in the list are forbidden (i.e. the moves that are leading back to the local minimum just visited).

The simplest way to run the tabu list is as a queue of fixed size \( k \); that is, when a new move is added to the list, the oldest move is discarded.

There is a more general mechanism that assigns a random number to each move that enters the list, ranging from \( k_{\text{min}} \) to \( k_{\text{max}} \) (the values \( k_{\text{min}} \) and \( k_{\text{max}} \) are parameters of the method), which represents
the number of iterations that should be kept in the tabu list. A move is removed from the list when its tabu period has expired. In this way the size on the list is not fixed, but varies dynamically in the interval $k_{\text{min}}$–$k_{\text{max}}$.

There is also a mechanism, called aspiration, that overrides the tabu status: if a move $m$ leads to a state whose cost function value is better than the best value found so far, then its tabu status is dropped and the resulting state is accepted as the new current one.

As for hill climbing, the stop criterion is based on the so-called idle iterations: the search terminates when a given number of iterations has elapsed from the last (strict) improvement of the cost function.

Composite local search

One of the attractive properties of the local-search paradigm is that different techniques can be combined and alternated to give rise to complex algorithms. In particular, a simple mechanism for combining different local-search techniques and/or different neighborhood relations is what we call the token-ring strategy.

Given an initial state $s_0$ and a set of basic local-search techniques $t_1, \ldots, t_q$, which we call runners, the token-ring search makes a circular run of each $t_i$, always starting from the best solution found by the previous runner $t_{i-1}$ (or $t_q$ if $i = 1$). The full token-ring run stops when it performs a fixed number of rounds with no improvement by any technique, whereas the component runners $t_i$ stop according to their specific criteria.

The effectiveness of token-ring search for two runners, called tandem search, has been stressed by several authors (see [6]). In particular, when one of the two runners, say $t_2$, is not used with the aim of improving the cost function, but rather for diversifying the search region, this idea falls under the name of iterated local search (see, e.g., [7]). In this case, the run with $t_2$ is normally called the mutation operator or the kick move. For example, in a recent work [8] we employed the alternation of tabu search using a small neighborhood with hill climbing using a larger neighborhood for the solution of the high-school timetabling problem.

ARCHITECTURE

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

The classes in the framework are split into five categories, depending on the role they play in a local-search algorithm. We have identified the following sets of classes.

- **Data classes** store the basic data of the algorithm. They encode the states of the search space, the moves and the input/output data. These classes only have data members and no methods, except for those accessing their own data. They have no computing capabilities and, generally, no links (pointers) to other classes.
• **Helpers** perform actions related to some specific aspects of the search. For example, the **Neighborhood Explorer** is responsible for everything concerning the neighborhood: candidate move selection, update of the current state by executing a move, and so on. Different **Neighborhood Explorers** may be defined in the case of composite search, each one handling a specific neighborhood relation used by the algorithm.

Helpers cooperate among themselves. For example, the **Neighborhood Explorer** is not responsible for the move prohibition mechanisms (such as maintaining the tabu list). These tasks are delegated to another helper, namely the **Prohibition Manager**.

Helpers do not have their own internal data, but they work on the internal state of the runners that invoke them and interact with them through function parameters.

• **Runners** are the algorithmic core of the framework. They are responsible for performing a run of a local-search technique, starting from an initial state and leading to a final one. Each runner has many data objects that represent the state of the search (current state, best state, current move, number of iterations, etc.) and it maintains links to all the helpers, which are invoked for performing specific tasks on its own data. Examples of runners are **tabu search** and **simulated annealing**.

• **Solvers** control the search by generating the initial solutions and deciding how and in which sequence runners have to be activated (e.g. tandem, multistart, hybrid search). In addition, they communicate with the external environment, by obtaining the input and delivering the output. They are linked to one or more runners (for simple or composite search, respectively) and to some of the helpers.

• **Testers** represent a simple predefined interface to the user program. They can be used to help the developers in debugging their code, adjusting the techniques and tuning the parameters. Furthermore, testers provide some tools for the analysis of the algorithms. In particular, the user can employ them to instruct the system to perform massive batch experiments and to collect the results in an aggregated form.

Testers are not used whenever the program is embedded in a larger application or if the users develop an **ad hoc** interface for their programs. For this reason, we do not consider testers as core components of EASYLOCAL++, but as development/analysis utilities.

The main classes that compose the core of EASYLOCAL++ are depicted in Figure 1 using UML notation [9]. The data classes, shown in small dashed 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 name is in a normal font represent the interface of the framework with respect to the user of EASYLOCAL++ and are meant for direct derivation of the user's concrete classes. Conversely, classes in italic typeface are only used as base classes for the other EASYLOCAL++ classes.

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

Note that the use of template classes for input and output forces the client to define two specific classes for dealing with the input and output of the search procedure. This is a deliberate design decision that encourages the user to explicitly identify input and output data, rather than mixing them in one or more objects.

The methods of EASYLOCAL++ interface classes can, in turn, be split into three categories that we call **MustDef**, **MayRedef**, **NoRedef** functions, as we now describe.
Figure 1. EASYLOCAL++ main classes.

- **MustDef**: pure virtual C++ functions that correspond to problem-specific aspects of the algorithm; they must be defined by the user and they encode some problem-specific aspects.
- **MayRedef**: non-pure virtual C++ functions that come with a tentative definition. These functions may be redefined by the user in the case where the default version is not satisfactory for the problem at hand (see the examples in the case study). Thanks to the late binding mechanism for virtual functions, the program always invokes the user-defined version of the function.
- **NoRedef**: final (non-virtual) C++ functions that 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 the framework, the user has to define the data classes (i.e. the template instantiations), the derived classes for the helpers and at least one runner and one solver. Figure 2 shows an example of one step of this process.
The names drawn in the box ChangeColorExplorer are MustDef functions and they are defined in the subclass ChangeColorExplorer. Conversely, in the box NeighborhoodExplorer some MayRedef functions are reported, which need not be redefined. The classes GraphCol, Coloring and Recolor, defined by the user, instantiate the templates Input, State and Move, respectively.

Many of the framework classes have no MustDef functions; as a consequence, the corresponding user-defined subclasses comprise only the class constructor, which in C++ cannot be inherited. For all the user’s classes, EASYLOCAL++ 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 only has to fill in the empty MustDef functions. Hence, as discussed in the case study, very little code actually needs to be written by the user.

MAIN COMPONENTS

We now describe the classes that compose EASYLOCAL++ in more detail. For this purpose, we present the data classes, the helpers, the runners, the solvers and their interaction. In addition, we briefly discuss the use of the testers.
Data classes

The data classes are used for template instantiation and hence have no actual code. They serve to store the following information (the example refers to our case study, namely the well-known $k$-GRAPH-COLORING problem).

- **Input**: input data of the problem, e.g. an undirected graph $G$ and an upper bound $k$ on the number of colors. We assume that the colors are represented by the integers $0, 1, \ldots, k - 1$.
- **Output**: output to be delivered to the user, e.g. an assignment of colors to all the nodes of the graph.
- **State**: an element of the search space, e.g. a (possibly partial) function that maps the nodes of the graph into the set of colors.
- **Move**: a local move, e.g. a triple $(v, c_{\text{old}}, c_{\text{new}})$ representing the fact that the color assigned to node $v$ in the map is changing from $c_{\text{old}}$ to $c_{\text{new}}$.

In a few applications, the **State** and **Output** classes may coincide but, in general, the **search space**—which is explored by the algorithm—is only an indirect (not necessarily complete) representation of the **output space**—which is related to the problem specification.

As an example, for the FLOWSHOP scheduling problem [10, Problem SS14, p. 241], the search space can be restricted to the set of task permutations, whereas the output space is the set of schedules with their start and end times for all tasks. The mapping between the two spaces is achieved through the concept of a **left-justified** schedule, i.e. a schedule that respects the task permutation and in which each task is scheduled at its earliest starting time. A general result assures that no information is lost with this mapping.

**Helper classes**

**EASYLOCAL++** defines five kinds of helper classes. Helpers are not related hierarchically, but they are linked to runners and to each other through pointers. Helpers in **EASYLOCAL++** are the following (three of which are discussed in more detail below).

- **State Manager**: this is responsible for all the operations on the state that are independent of the neighborhood definition.
- **Output Producer**: this is responsible for translation between elements of the search space and output solutions. It also delivers the other output information of the search and stores and retrieves solutions from files. This is the only helper that deals with the **Output** class. All the other helpers work only on the **State** class, which represents the elements of the search space used by the algorithms.
- **Neighborhood Explorer**: this handles all the features concerning neighborhood exploration.
- **Prohibition Manager**: this is in charge of the management of the prohibition mechanism (e.g. for the tabu-search strategy).
- **Weight Handler**: this is responsible for the adaptive modification of the weights of the cost function, according to the shifting penalty mechanism (see, e.g. [11]). For each component of the cost function, it maintains an independent weight, which varies depending on the number of violations and according to a customizable scheme.

Only the **State Manager**, the **Output Producer**, and the **Neighborhood Explorer** are common to all algorithms whereas the other two are only used by some specific techniques.
State Manager

The State Manager is responsible for all the operations on the state that are independent of the neighborhood definition; therefore, no Move definition is supplied to the State Manager. Its core functions are as follows.

- **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;
  - Violations(const State &st): counts the number of violated constraints in the state st.

- **MayRedef functions:**
  - SampleState(State &st, int n): stores in st the best among n randomly generated states;
  - BuildState(State &st): generates a state according to some problem-specific algorithm and stores it in st. Its tentative definition simply calls RandomState(st).

Neighborhood Explorer

A Neighborhood Explorer encodes a particular neighborhood relation associated with a specific Move class; therefore, if different neighborhood relations are used (e.g. in the token-ring strategy) different subclasses of NeighborhoodExplorer with different instantiations for the template Move must be defined.

Some of the main functions of the Neighborhood Explorer are as follows.

- **MustDef functions:**
  - MakeMove(State &st, const Move &mv): updates the state st by applying the move mv to it;
  - RandomMove(const State &st, Move &mv): generates a random move for the state st and stores 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 function is used in algorithms relying on exhaustive neighborhood exploration.

- **MayRedef functions:**
  - DeltaObjective(const State &st, const Move &mv): computes the difference in the objective function (soft constraints) between the state obtained from st applying mv and the state st itself;
  - DeltaViolations(const State &st, const Move &mv): computes the difference in the violations count (hard constraints) between the state obtained from st applying mv and the state st itself;
- \texttt{DeltaCostFunction(const State \&st, const Move \&mv)}: computes a weighted sum of \texttt{DeltaObjective(...)} and \texttt{DeltaViolations(...)};
- \texttt{FirstMove(const State \&st, Move \&mv)}: generates the first move for the state \texttt{st} according to the neighborhood exploration strategy and stores it in \texttt{mv}. Its tentative definition simply invokes the \texttt{RandomMove} method;
- \texttt{BestMove(const State \&st, Move \&mv)}: computes the best possible move in the neighborhood of \texttt{st};
- \texttt{SampleMove(const State \&st, Move \&mv, int n)}: computes the best among \texttt{n} random neighbors of \texttt{st};
- \texttt{BestNonProhibitedMove(const State \&st, Move \&mv, ...)}: computes the best move not in the prohibition set. A check of the prohibited status is done by a call to the associated \textbf{Prohibition Manager}. This function has other parameters (here omitted) that are passed to the \textbf{Prohibition Manager}.

Note that the two functions \texttt{DeltaObjective()} and \texttt{DeltaViolations()} are listed within the \textbf{MayRedef} because they have a tentative definition. However, their tentative definition corresponds to explicitly computing $f(s \oplus m)$ and $f(s)$, by invoking the corresponding methods of the \textbf{State Manager}, and returning the difference. These definitions are unacceptably inefficient for almost all applications and they should be redefined only taking into account the differences generated by the local changes.

As an example of \texttt{EASYLOCAL++} code, we present the definition of the \texttt{BestMove()} function. In the following code the type \texttt{fvalue} denotes the codomain of the objective function (typically \texttt{int} or \texttt{double}) and \texttt{LastMoveDone()} is a \textbf{MayRedef} function whose tentative code is the single instruction `'return mv == start move;'`.

```cpp
template <class Input, class State, class Move>
fvalue NeighborhoodExplorer<Input,State,Move>::BestMove(const State &st,
Move &mv)
{
    FirstMove(st, mv);
    fvalue mv_cost = DeltaCostFunction(st, mv);
    best_move = mv;
    fvalue best_delta = mv_cost;
    do
    {
        mv_cost = DeltaCostFunction(st, mv);
        if (mv_cost < best_delta)
        {
            best_move = mv;
            best_delta = mv_cost;
        }
        NextMove(st, mv);
    } while (!LastMoveDone(st, mv));
    mv = best_move;
    return best_delta;
}
```
The Neighborhood Explorer also includes functions for more sophisticated selection mechanisms. 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.

Note that the computation of the cost function is done partly by the neighborhood explorer, which computes the variations and partly by the State Manager that computes the static value. This design choice is due to the fact that the variation of the cost function is dependent on the neighborhood relation and different Neighborhood Explorers compute the variations differently. In this way, we can add new neighborhood definitions without changing the State Manager.

Prohibition Manager

This helper deals with move prohibition mechanisms that prevent cycling and allow for diversification. As shown in Figure 1, we also have a more specific one, which maintains a list of Move elements according to the prohibition mechanisms of the tabu search. Its main functions are as follows.

- **MustDef functions:**
  - `Inverse(const Move &m1, const Move &m2)`: checks whether a (candidate) move m1 is the inverse of a (list member) move m2.

- **MayRedef functions:**
  - `InsertMove(const Move &mv, ...)`: inserts the move mv in the list and assigns it a tenure period; furthermore, it discards all moves whose tenure period is expired;
  - `ProhibitedMove(const Move &mv, ...)`: checks whether a move is prohibited, i.e. whether it is the inverse of one of the moves in the list.

Both functions `InsertMove()` and `ProhibitedMove()` have other parameters, which are related to the aspiration mechanism of tabu search that is not described here.

Runners

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

The class MoveRunner also requires the template Move and the pointers to the necessary helpers. 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 directly define objects of the type State, such as `current_state` and `best_state`, rather than accessing them through pointers. This feature makes the construction and copying of objects of the type State completely transparent to the user, since they do not require any explicit cast operation or dynamic allocation.

We present `Go()`, the main function of MoveRunner, which performs a full run of a local search.
template <class Input, class State, class 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 abstract methods that will be defined in the subclasses of MoveRunner. For example, if we call p_nhe the pointer to the Neighborhood Explorer, the SelectMove() function invokes p_nhe->RandomMove() in the subclass SimulatedAnnealing, whilst in the subclass TabuSearch it invokes p_nhe->BestNon-ProhibitedMove().

Two functions, which are defined at this level of the hierarchy, are the MayRedef functions UpdateIterationCounter() and LowerBoundReached(). Their tentative definition simply consists of incrementing the iteration counter by one and in checking if the current state cost is equal to zero, respectively.

Among the actual runners, TabuSearch is the most complex. This class has extra data for the specific features of tabu search. Its extra members include:

- a State variable for the best state, which is necessary since the search can go uphill;
- a pointer to the Prohibition Manager, which is used by the functions SelectMove() and StoreMove();
- two integer variables iteration_of_best and max_idle_iterations for implementing the stop criterion.

We also provide an advanced version of tabu search that includes the shifting penalty mechanism. The corresponding class then works in connection with a Weight Handler, which implements the chosen adaptive weighting strategy.

Solvers

Solvers represent the external layer of EASYLOCAL++. 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 as follows.

- MayRedef functions:
  - FindInitialState(): gets the initial state by calling the function Sample-State() of the helper State Manager on the internal state of the solver;
Run(): starts the local-search process, invoking the Go() function of the runners according to the solver strategy.

• NoRedef functions:

  - Solve(): makes a complete execution of the solver, by invoking the functions FindInitialState(), Run() and DeliverOutput();
  - MultiStartSolve(): makes many runs from different initial states and delivers the best of all the final states as its output;
  - DeliverOutput(): calls the function OutputState() of the helper OutputProducer on the internal state of the solver;
  - AddRunner(Runner *r): for the SimpleSolver, it replaces the current runner with r, whilst for MultiRunnerSolver it adds r to the bottom of its list of runners;
  - ClearRunners(): removes all runners attached to the solver.

Various solvers differ mainly by their definition of the Run() function. For example, for TokenRingSolver, which manages a pool of runners, it consists of a circular invocation of the Go() function for each runner. Similarly, for the ComparativeSolver, the function Go() of all the runners is invoked on the same initial state and the best outcome becomes the new internal state of the solver.

The core of the function Run() of TokenRingSolver is given below. The solver’s variable internal_state is previously set to the initial state by the function FindInitialState().

```cpp
template <class Input, class Output, class State>
void TokenRingSolver<Input, Output, State>::Run()
{
    
    current_runner = 0;
    previous_runner = runners_no;

    
    runners[current_runner]->SetCurrentState(internal_state);
    while (idle_rounds < max_idle_rounds && !interrupt_search)
    {
        
        do
        {
            runners[current_runner]->Go(); // let current runner go()
            total_iterations += runners[current_runner]->NumberOfIterations();
            if (runners[current_runner]->BestStateCost() < internal_state_cost)
            {
                internal_state = runners[current_runner]->GetBestState();
                internal_state_cost = runners[current_runner]->BestStateCost();
                if (runners[current_runner]->LowerBoundReached())
                {
                    interrupt_search = true;
                    break;
                }
                else
                    improvement_found = true;
            }
        }
    }
```
Note that both solvers and runners have their own state variables and they communicate through the functions GetCurrentState() and SetCurrentState(). This feature is used, for instance, by the comparative solver, which makes a run of all runners and updates its internal state with the final state of the runner that has given the best result.

Testers

Testers represent a text-based user interface to the program. They support both interactive and batch runs of the system, collecting data for the analysis of the algorithms.

For the interactive run, a tester allows the user to perform runs of any of the available runners and keeps track of the evolution of the current state. If requested, for debugging purposes, runs can be fully traced to a log file. At any moment, the user can check the current violations and objective, and to store/retrieve the current state in/from data files.

A specialized tester class, called MoveTester, is used to perform single moves one at a time. The user specifies the neighborhood relation to be used and the move strategy (best, random, from input, etc.) and the system returns the selected move, together with all corresponding information about the variation of the cost function. In addition, a MoveTester provides various auxiliary functions, such as checking the cardinality of the neighborhood.

Finally, there is a specific tester for running experiments in unsupervised mode. This tester accepts experiment specifications in a given language, called EXPSPEC, and executes all of them sequentially. The language is a straightforward script language, used for instructing massive batch experiments with different parameter settings. This is particularly useful for tuning local-search techniques, since most of the time their success is dependent on a careful choice of parameters.

The syntax of the EXPSPEC language has been kept as small as possible in order to minimize the demand on the user to learn another language. A statement of the language specifies an experiment on a given instance in terms of the solving strategy to be employed (i.e. the runners to be activated and their parameter settings). Furthermore, it is possible to specify how many runs of the algorithms should be performed and where the data collected in the experiments should be stored.
As an example of EXPSPEC code consider the file:

Instance "DSJC250.1.col:9"
{
  Trials: 10;
  Runner tabu search "Exhaustive Explorer"
  {
    min tabu tenure: 10;
    max tabu tenure: 20;
    max idle iterations: 1000;
    max iterations: 100000;
  }
  Runner hill climbing "Double Climber"
  {
    max idle iterations: 1000;
  }
}

In this example, the tester performs 10 runs on the instance DSJC125.1.col of a tandem composed of the tabu search runner Exhaustive Explorer and the hill-climbing runner Double Climber, with the parameter settings provided in the file (enclosed in curly brackets).
The tester also collects statistical data on the solutions found, which is shown to the user in aggregated graphical form. The information collected by the tester allows the user to analyze and compare different algorithms and/or different parameter settings on the same instances of the problem, with very little intervention by the human operator.

Furthermore, a tester automatically produces plots like that reported in Figure 3. The plot shows the value of the cost of the current state for an individual run. These plots give a qualitative view of the behavior of the algorithm and help the algorithm designer to improve its features and settings.

The testers are implemented as concrete classes that can be used directly, with no need to define derived classes. The EXPSPEC interpreter has been written using Flex and Bison [12,13] and can be easily customized by an expert user if necessary.

A CASE STUDY: THE $k$-GRAPH COLORING PROBLEM

As an example of the actual use of EASYLOCAL++, we present the development of local-search algorithms for the $k$-GRAPH COLORING problem [10, Problem GT4, p. 191]. The statement of the problem is the following: we are given an undirected graph $G = (V, E)$ and a set of $k$ colors $C = \{0, 1, \ldots, k - 1\}$. The problem is to assign to each node $v \in V$ a color value $c(v) \in C$ such that adjacent nodes are assigned different colors (i.e. for all $(v, w) \in E$ $c(v) \neq c(w)$).

Data classes

In this section we define the classes that represent the problem-specific part of the algorithm, namely: the input, the output, the search space and the moves. They will be used for template instantiation.

Input

The input of the problem is a graph and an upper bound on the number of colors that can be used to color its vertices. With the aim of representing the graph we exploit the class `leda_ugraph`, which is available in the LEDA library [1]. Hence, to instantiate the template `Input`, we define a class that inherits from `leda_ugraph` and we add to it an integer $k$, which represent the upper bound on the number of colors. The resulting class declaration is as follows:

```cpp
class Graph
    : public leda_ugraph
{
public:
    unsigned int k;
    void Load(string id);
};
```

The method `Load()` instantiates the graph by loading it from a file in the DIMACS format [14].

Output

The output of the problem is a function $c : V \to C$ from graph nodes to color values. We use a map from nodes to colors provided by the LEDA library to instantiate the template `Output`. The class, reported below, also includes a pointer to the input class that is needed to initialize the map.
class GraphColoring
 : public leda_node_map<Color>
{
public:
    GraphColoring()
      : p_in(NULL) {}
    GraphColoring(Graph *g)
      : p_in(g)
      { init(*g); }
    void SetInput(Graph *g)
      { p_in = g; init(*g); }
protected:
    Graph *p_in;
};

The class GraphColoring has a constructor that takes as its argument a pointer to a Graph object, which initializes the object based on the information contained in the graph. In addition, it has a constructor with no arguments that leaves the object uninitialized, and a function SetInput(), which initializes (or reinitializes) an already existing object according to the provided input.

Such functions, namely the two constructors and SetInput(), are the only mandatory members for a class that instantiates the Output template and EASYLOCAL++ relies on their presence.

Search space

The search space of our algorithms is the set of all possible colorings including the infeasible ones. Therefore, we choose to instantiate the template State with the class Coloring that is also a map from graph nodes to color values. However, differently from the output class, from which it is derived, the state class includes redundant data structures used for efficiency purposes. In particular, it includes a set, called conflicts, that contains all conflicting nodes, i.e. the nodes that have at least one adjacent node with the same color.

class Coloring
 : public GraphColoring
{
public:
    Coloring()
      : GraphColoring()
      { conflicts.clear(); }
    Coloring(Graph *g)
      : GraphColoring(g)
      { conflicts.clear(); }
    void SetInput(Graph *g)
      { GraphColoring::SetInput(g); conflicts.clear(); }
    leda_set<leda_node> conflicts;
};

Similarly to the Output, for the State class the default constructor, the constructor that receives a pointer to the Input class, and the function SetInput() are mandatory.
Move

The first neighborhood relation that we consider is defined by the change in the color of one conflicting node. Hence, a move can be identified by a triple \( \langle v, c_{\text{old}}, c_{\text{new}} \rangle \) composed of the node \( v \), its current color \( c_{\text{old}} \) and the newly assigned color \( c_{\text{new}} \). For implementing this kind of move, we define a class, called Recolor, as follows:

```cpp
class Recolor
{
public:
  leda_node v;
  color c_new, c_old;
};
```

Note that in order to select and apply a move \( m \) from a given state \( s \) we only need the node \( v \) and the new color \( c_{\text{new}} \). Nevertheless, it is also necessary to store the old color for the management of the prohibition mechanisms. In fact, the tabu list only stores the ‘raw’ moves regardless of the states in which they have been applied. In addition, the presence of the data member \( c_{\text{old}} \) makes the code simpler and slightly improves the efficiency of various functions.

Helpers

We have to define four helpers, namely State Manager, Output Producer, Neighborhood Explorer and Prohibition Manager, which encode some problem-specific features associated with different aspects of the search.

State Manager

We start by describing the State Manager that handles the Coloring state and is represented by the following class:

```cpp
class ColoringManager
  : public StateManager<Graph,Coloring>
{
public:
  ColoringManager(Graph *g = NULL)
    : StateManager<Graph,Coloring>(g) {}
  void RandomState(Coloring &);
protected:
  void SetInput(Graph *g)
  { p_in = g; }
  fvalue Violations(const Coloring &)
    { return 0; } // no objective for this problem
};
```

Copyright © 2003 John Wiley & Sons, Ltd.  
Softw. Pract. Exper. 2003; 33:733–765
The only two functions that need to be defined are RandomState() and Violations(), given that the others have been defined inline.

The function RandomState() assigns a random color to each node and builds the conflict set accordingly.

```c++
void ColoringManager::RandomState(Coloring &col)
{
    leda_node v, w;
    leda_edge e;

    forall_nodes(v, *p_in) // for each node v in the graph
    { col[v] = Random(0, p_in->k - 1); } // assign a random color in [0, k-1]

    forall_edges(e, *p_in) // rebuild the conflict set
    { // v and w are adjacent in the graph
        v = p_in->source(e);
        w = p_in->target(e);
        if (col[v] == col[w]) // if their color is the same
        { // insert them in the conflict set
            col.conflicts.insert(v);
            col.conflicts.insert(w);
        }
    }
}
```

The function Violations() simply counts the conflicting edges by means of the following code:

```c++
fvalue ColoringManager::Violations(const Coloring &col) const
{
    fvalue viol = 0;
    leda_edge e;
    leda_node v, w;

    forall_edges(e, *p_in) // for each edge v, w in the graph
    { // if the nodes have the same color
        v = p_in->source(e);
        w = p_in->target(e);
        if (col[v] == col[w]) // a new violation has found
        { viol++;
        }
    }
    return viol;
}
```

Neighborhood Explorer

We now move to the description of the Neighborhood Explorer for the Recolor move, which is represented by the class RecolorExplorer, defined as follows:
class RecolorExplorer
  : public NeighborhoodExplorer<Graph,Coloring,Recolor>
{
public:
  RecolorExplorer(StateManager<Graph,Coloring> *psm, Graph *pin)
    : NeighborhoodExplorer<Graph,Coloring,Recolor>(psm,pin)
  {};
  void FirstMove(const Coloring &, Recolor &);
  void RandomMove(const Coloring &, Recolor &);
  void MakeMove(Coloring &col, const Recolor &rc);
protected:
  bool LastMoveDone(const Recolor &);
  fvalue DeltaViolations(const Coloring &, const Recolor &);
  fvalue DeltaObjective(const Coloring &, const Recolor &)
    { return 0; }
  void NextMove(const Coloring &, Recolor &);
private:
  leda_list<leda_node> candidate_nodes;
};

The only data member is a list of nodes, called candidate_nodes, which is used for the systematic exploration performed by the three functions FirstMove(), NextMove() and LastMoveDone(). Specifically, this list is initialized to be equal to the conflict set of the current state by the function FirstMove(), it is emptied by the function NextMove() as it processes the nodes, and it is checked for emptiness by the function LastMoveDone().

Among these three functions, we show the implementation of the most interesting one, namely NextMove(). This function assigns to \( c_{\text{new}} \) the successive value (modulo \( k \)); if \( c_{\text{new}} \) is equal to \( c_{\text{old}} \), the exploration for that node is finished and the next node in the list candidate_nodes is processed.

void RecolorExplorer::NextMove(const Coloring &col, Recolor &rc)
{
  rc.c_new = (rc.c_new + 1) % p_in->k; // try the next color
  if (rc.c_new == rc.c_old && !candidate_nodes.empty()) // if the color exploration for
    rc.v = candidate_nodes.pop(); // start with a new node
    rc.c_old = col[rc.v];
    rc.c_new = (rc.c_old + 1) % p_in->k;
}

The situation when candidate_nodes is empty and \( c_{\text{new}} \) is equal to \( c_{\text{old}} \) is detected by LastMoveDone() which returns true and stops the search.

We now show the function RandomMove(), which simply picks a random node from the conflict set and picks a new random color for it.
The function\texttt{DeltaViolations()} computes the difference between the number of the nodes adjacent to \texttt{rc.v} colored with \texttt{c_new} and those colored with \texttt{c_old}.

\begin{verbatim}
// DeltaViolations()
fvalue RecolorExplorer::DeltaViolations(const Coloring &col, const Recolor &rc) {
    fvalue delta = 0;
    leda_node w;
    forall_adj_nodes(w, rc.v) // for all the nodes w adjacent to v
    {
        if (col[w] == rc.c_new) // if the color is the same as c_new
            delta++; // a new conflict would be added
        else if (col[w] == rc.c_old) // if the color is the same as c_old
            delta--; // an old conflict would be removed
    }
    return delta;
}
\end{verbatim}

This function checks each node adjacent to \texttt{rc.v} and detects whether it is involved in a new conflict or if an old conflict has been removed by the new assignment. The adjacent nodes are generated by means of the \texttt{for\_all\_adj\_nodes} statement of LEDA.

Finally, the \texttt{MakeMove} function updates the color of the node \texttt{rc.v} to the new value and recomputes the set of conflicting nodes by inspecting all the nodes that are adjacent to \texttt{rc.v}.

\begin{verbatim}
// MakeMove()
void RecolorExplorer::MakeMove(Coloring &col, const Recolor &rc) {
    bool rc_v_in_conflict = false;
    leda_set<leda_node> check_for_removal;
    leda_node w;
    // first, update the color of rc.v
    col[rc.v] = rc.c_new;
    // then, check the adjacent nodes for conflicts addition/removal
    forall_adj_nodes(w, rc.v) // for all the nodes w adjacent to rc.v
    {
        if (col[w] == rc.c_new) // if the color is the same as c_new
            // a new conflict would be added
        else if (col[w] == rc.c_old) // if the color is the same as c_old
            // an old conflict would be removed
    }
}
\end{verbatim}
{  
    rc_v_in_conflict = true;  // then rc.v is still in conflict
    col.conflicts.insert(w);  // and possibly a new conflict is added
  }
else if (col[w] == rc.c_old)  // if the color is the same as c_old
  check_for_removal.insert(w);  // an old conflict could be removed

  // if the move has not added conflicts
  // we can safely remove rc.v from the conflict set
  if (!rc_v_in_conflict)
    col.conflicts.del(rc.v);

  // at last we must check the nodes previously in conflict with rc.v
  // for a possible removal from the conflict set
  forall(w, check_for_removal)
  {
    // to this aim, we check whether that nodes are still in conflict
    // with some other node, or they can be safely removed
    bool w_still_in_conflict = false;
    forall_adj_nodes(u, w)
    {
      if (col[w] == col[u])  // w is still in conflict
        w_still_in_conflict = true;  // with at least another node
    }
    if (!w_still_in_conflict)  // if w is not more in conflict
      col.conflicts.del(w);  // it can be removed
  }
}

Prohibition Manager

The Prohibition Manager for this problem is provided by the class ProhibitedColorsManager. The full code of the class, which consists of a constructor and of the only MustDef function Inverse(), is included within the class definition reported below.

class ProhibitedColorsManager
  : public TabuListManager<Recolor>
{
  public:
    ProhibitedColorsManager(int min = 0, int max = 0)
      : TabuListManager<Recolor>(min,max)
    {
    }
  protected:
    bool Inverse(const Recolor &rc1, const Recolor &rc2) const
    {
      return (rc1.v == rc2.v &&
               (rc1.c_new == rc2.c_old || rc2.c_new == rc1.c_old));
    }
};

Note that according to the above definition of the function Inverse(), we consider a move inverse to another one if it involves the same node and it has one of the two colors in common.
Double moves

As explained in the next section, our algorithms make use also of a more complex neighborhood relation. Such a relation is defined by the so-called double moves, which are made by a pair of Recolor moves. In more detail, a double move consists of recoloring one conflicting node and one from its adjacency. For this neighborhood, we define the Neighborhood Explorer and Prohibition Manager classes and the suitable data class DoubleRecolor for instantiating the template move. The definition of these classes falls outside the scope of this case study.

Runners

We define six runners. Three are the straightforward implementations of the three basic techniques using the Recolor move. No function needs to be defined for these runners and their code only results in a template instantiation. For example, the definition of the hill-climbing runner is the following.

```cpp
class HillClimbingColoring : public HillClimbing<Graph,Coloring,Recolor>
{
public:
    HillClimbingColoring(StateManager<Graph,Coloring> *psm,
                         NeighborhoodExplorer<Graph,Coloring,Recolor> *pnhe,
                         Graph *g = NULL)
        : HillClimbing<Graph,Coloring,Recolor>(psm,pnhe,g)
    {}
};
```

This definition is included entirely in the skeleton code provided by EASYLOCAL++. In this case the user needs only to supply the name of the problem-specific classes.

The fourth runner is a variant of tabu search that uses a non-systematic exploration of the neighborhood. The implementation of this runner is achieved by redefining the MayRedef function SelectMove() so that it invokes a function of the Neighborhood Explorer called SampleNonProhibitedMove(), rather than the function BestNonProhibitedMove() (as in the tentative code). Its class definition is:

```cpp
class SampleTabuColoring : public TabuSearch<Graph,Coloring,Recolor>
{
public:
    SampleTabuColoring(StateManager<Graph,Coloring> *psm,
                        NeighborhoodExplorer<Graph,Coloring,Recolor> *pnhe,
                        TabuListManager<Recolor> *ptlm, int samp,
                        Graph *g = NULL)
        : TabuSearch<Graph,Coloring,Recolor>(psm,pnhe,ptlm,g), samples(samp)
    {}

protected:
    void SelectMove()
    {
        current_move_cost = p_nhe->SampleNonTabuMove(current_state,
```
The fifth and the sixth runners implement a tabu search using DoubleRecolor moves, based on the best (DoubleTabuColoring) and sample moves (SampleDoubleTabuColoring), respectively.

The selection based on the sample moves can again be obtained by redefining the function SelectMove() as shown above for the SampleTabuColoring class. Nevertheless, for the purpose of code reuse, we can go a step forward and define a template superclass SampleTabuSearch from which the classes SampleTabuColoring and SampleDoubleTabuColoring can be derived. In this way, the code written for this new strategy can be exploited in the solution of other problems, which may be very different from the problem at hand. The code of the template class is the following.

```
template <class Input, class Output, class Move>
class SampleTabuSearch<Input,Output,Move>
  : public TabuSearch<Input,Output,Move>
{
  public:
    SampleTabuSearch(StateManager<Input,State> *psm,
    NeighborhoodExplorer<Input,State,Move> *pnh,
    TabuListManager<Move> *ptlm, int samp, Input *pin = NULL)
    : TabuSearch<Input,State,Move>(psm,pnh,ptlm,pin), samples(samp)
  {
  }

protected:
  void SelectMove()
  {
      current_move_cost = p_nhe->SampleNonTabuMove(current_state,
            current_move, samples, current_state_cost,
            best_state_cost);
  }
}
```

Given this class, the above-mentioned runners can then be obtained simply by instantiating the templates Input and State with the classes Graph and Coloring, and by instantiating the template Move with the class Recolor or with the class DoubleRecolor.

**Solvers**

We define simple and token-ring solvers. The first is used for running the basic techniques. The solver can run different techniques by changing the runner attached to it by means of the function AddRunner(). The latter solver is used for running various tandems of two runners. The runners participating in the tandem are simply selected using AddRunner() and ClearRunners(), and the composition does not require any other additional programming effort.

Similarly to the first three runners, the solvers’ derivation is only a template instantiation and, as in the previous case, this operation is fully supported by the skeleton code.
Table I. Performances of simple solvers.

<table>
<thead>
<tr>
<th>Instance k</th>
<th>HC</th>
<th>SA</th>
<th>TS</th>
<th>TSs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T V S</td>
<td>T V S</td>
<td>T V S</td>
<td>T V S</td>
</tr>
<tr>
<td>125.1</td>
<td>6 30.8 1.2 0</td>
<td>0.39 0.0</td>
<td>0.03 0.0</td>
<td>0.07 0.0</td>
</tr>
<tr>
<td>125.5</td>
<td>18 14.4 4.9 0</td>
<td>65.8 2.0</td>
<td>2.65 0.0</td>
<td>15.2 0.8</td>
</tr>
<tr>
<td>125.9</td>
<td>44 26.5 2.1 0</td>
<td>537 0.0 8</td>
<td>13.5 0.0</td>
<td>42.5 0.1</td>
</tr>
<tr>
<td>250.1</td>
<td>9 34.6 4.3 0</td>
<td>15.8 0.0</td>
<td>0.61 0.0</td>
<td>3.58 0.0</td>
</tr>
<tr>
<td>250.5</td>
<td>30 28.2 10.3 0</td>
<td>459 3.5 0</td>
<td>52.2 0.0</td>
<td>169 1.3</td>
</tr>
<tr>
<td>250.9</td>
<td>75 68.3 3.5 0</td>
<td>1452 0.0</td>
<td>118 0.0</td>
<td>173 0.0</td>
</tr>
<tr>
<td>500.1</td>
<td>14 85.7 6.1 0</td>
<td>74.3 0.0</td>
<td>7.05 0.0</td>
<td>26.9 0.0</td>
</tr>
<tr>
<td>500.5</td>
<td>54 69.6 16.7 0</td>
<td>1506 0.5</td>
<td>457 0.0</td>
<td>1069 1.7</td>
</tr>
<tr>
<td>500.9</td>
<td>140 194 6.4 0</td>
<td>1370 1.0</td>
<td>1856 0.0</td>
<td>2848 0.0</td>
</tr>
</tbody>
</table>

Experimental results

The described GRAPHCOLORING implementation is composed of about 1000 lines of C++ code for all the implemented techniques. However the real programming effort (i.e. not taking into account the skeleton code) consists of about 700 lines of code.

For the purpose of evaluating the algorithms, we have run them on a set of nine instances taken from the DIMACS benchmark repository. In particular, we select the family of random graphs denoted by the prefix DSJC proposed by Johnson et al. [15]. For each instance we have performed 10 runs recording the best solution of each run. The experiments have been performed on a 650 MHz PC running Linux. Both the framework and the case-study files have been compiled with the GNU C++ compiler release 2.95.2 turning on the -O3 optimization flag.

The results of the basic and tandem techniques are summarized in Tables I and II, respectively. The names in the tables should be read as follows: HC (hill climbing), SA (simulated annealing) and TS (tabu search) denote the basic techniques with the Recolor neighborhood relation. The subscript s means that the runner uses sampling neighborhood exploration and the superscript 2 means that it uses the DoubleRecolor neighborhood. Finally, tandem search is identified by the symbol + between the two runners. For each solver, there are three columns, which have the following meanings:

- T, average running time (times are expressed in seconds);
- V, average number of violations;
- S, number of trials that reached a solution with no violations.

Table I shows quite clearly that (for this set of instances) TS is superior to the other techniques. In addition, Table II shows that there is not much to gain by using double moves and tandems of runners.

However, for the sake of fairness, we should take into account the fact that we have not performed a full parameter tuning session; parameters have been set to suitable values according to the literature. Nevertheless, drawing conclusions about the k-GRAPHCOLORING problem is not the focus of this
Table II. Performances of solvers using double moves and tandem solvers.

<table>
<thead>
<tr>
<th>Instance</th>
<th>k</th>
<th>TS$^2$</th>
<th>TS$^2$</th>
<th>HC + TS$^2$</th>
<th>TS + TS$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T</td>
<td>V</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>125.1</td>
<td>6</td>
<td>0.431</td>
<td>0.0</td>
<td>10</td>
<td>11.9</td>
</tr>
<tr>
<td>125.5</td>
<td>18</td>
<td>72.7</td>
<td>0.0</td>
<td>10</td>
<td>30.3</td>
</tr>
<tr>
<td>125.9</td>
<td>44</td>
<td>1619</td>
<td>0.0</td>
<td>10</td>
<td>123</td>
</tr>
<tr>
<td>250.1</td>
<td>9</td>
<td>12.4</td>
<td>0.0</td>
<td>10</td>
<td>80.8</td>
</tr>
<tr>
<td>250.5</td>
<td>30</td>
<td>2775</td>
<td>0.0</td>
<td>10</td>
<td>310</td>
</tr>
<tr>
<td>250.9</td>
<td>75</td>
<td>2448</td>
<td>0.0</td>
<td>10</td>
<td>501</td>
</tr>
<tr>
<td>500.1</td>
<td>14</td>
<td>253</td>
<td>0.0</td>
<td>10</td>
<td>278</td>
</tr>
<tr>
<td>500.5</td>
<td>54</td>
<td>2220</td>
<td>0.0</td>
<td>10</td>
<td>1604</td>
</tr>
<tr>
<td>500.9</td>
<td>140</td>
<td>8146</td>
<td>0.0</td>
<td>10</td>
<td>1076</td>
</tr>
</tbody>
</table>

Table III. Comparison with a direct implementation of the tabu-search solver.

<table>
<thead>
<tr>
<th>Instance</th>
<th>k</th>
<th>Optimization disabled</th>
<th>Optimization enabled (~O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T_T</td>
<td>T_d</td>
</tr>
<tr>
<td>125.1</td>
<td>6</td>
<td>0.14</td>
<td>0.13</td>
</tr>
<tr>
<td>125.5</td>
<td>18</td>
<td>14.5</td>
<td>13.7</td>
</tr>
<tr>
<td>125.9</td>
<td>44</td>
<td>113</td>
<td>106</td>
</tr>
<tr>
<td>250.1</td>
<td>9</td>
<td>3.11</td>
<td>3.05</td>
</tr>
<tr>
<td>250.5</td>
<td>30</td>
<td>293</td>
<td>283</td>
</tr>
<tr>
<td>250.9</td>
<td>75</td>
<td>779</td>
<td>741</td>
</tr>
<tr>
<td>500.1</td>
<td>14</td>
<td>36.6</td>
<td>35.0</td>
</tr>
<tr>
<td>500.5</td>
<td>54</td>
<td>1962</td>
<td>1854</td>
</tr>
<tr>
<td>500.9</td>
<td>140</td>
<td>7695</td>
<td>7387</td>
</tr>
</tbody>
</table>

Our goal is to show the usefulness of these tables, which can be obtained with limited effort using the ExpSpec language of EasyLocal++.

For the purpose of measuring the overhead introduced by the framework, we developed a companion plain C++ implementation of our tabu-search solver. The differences with respect to the EasyLocal++ implementation reside in the fact that the local-search code does not use virtual functions and the function calls are optimized by inline expansion. The implementation consists of 550 lines of code and it relies on the same data structures as those employed in EasyLocal++, in order to obtain a fair comparison. Note that the amount of code needed to write a local-search solver from scratch is comparable to the amount of code written for developing a whole family of solvers using EasyLocal++.
We measure the performances of the implementations in two different settings. First we compile the programs without any optimization and we run the whole series of experiments on the test-bed. Then, we turn on the `-O3` compiler optimization flag and we perform the experiments again.

The data collected in the experiences are presented in Table III. We denote with $T_{el}$ the running times of the EASY LOCAL ++ implementation, whereas with $T_{d}$ we refer to the running times of the plain C++ solver. Moreover, we use the superscript o to indicate the optimized versions. In the third column of each set of experiments we report the performance loss of the EASY LOCAL ++ implementation. This measure is computed as the ratio between the difference of the running times of the two implementations and the running time of the direct implementation.

The table shows that the behavior of the two implementations is similar. The performance loss is about 5% if code optimization is disabled, whereas it is about 10% if the executable is fully optimized. Moreover, one can also notice that the ‘gap’ between the two implementations becomes smaller for higher running times and that the behavior of the non-optimized solvers is more stable with respect to the optimized versions.

Although the performance loss of the optimized EASY LOCAL ++ implementation is not negligible, this is the typical degradation of programs that make extensive use of virtual functions and is, therefore, unavoidable for this type of framework. We believe that this is an acceptable drawback compared with its advantages.

**RELATED WORK**

The idea of illustrating local-search techniques by means of generic algorithms has been proposed [16] and [17] among others.

In [16] the use of a local-search template to classify existing local-search techniques has been proposed and new types of search algorithm belonging to the local-search family are suggested. In [17] a conceptual framework for local search is described. This framework differs from the first one since, similarly to EASY LOCAL ++, it relies on design patterns. In addition, the authors discuss in detail a constructive search phase used for finding the initial state for local search.

More interesting for our comparison are the software systems that actively support the design and implementation of algorithms.

**Black-box systems and toolkits**

One class of available software tools are the black-box systems. The main difference of such systems with respect to EASY LOCAL ++ is that the program corresponding to the specified algorithm is assembled internally by the system and, therefore, its execution is performed 'behind the scenes'. Conversely, EASY LOCAL ++ is completely glass-box (or white-box), and the exact C++ statements of the program are ‘before the user’s eyes’. Furthermore, in most black-box systems users must learn the syntax of the specification language from scratch. In EASY LOCAL ++, instead, the algorithm is written in a language (i.e. C++) that a skilled user might already know, or at least be well-disposed to learn. In addition, the interface with external modules and public libraries, which is crucial for industrial applications, might be quite cumbersome in black-box systems. Finally, EASY LOCAL ++ is fully extensible and customizable by the expert user by means of new derived
classes. In contrast, the modification of the black-box system would require the intervention of the system designers.

Examples of remarkable black-box systems are Localizer [18] and SALSA [19]. The latter is a language for general search algorithms (exhaustive and non-exhaustive) and has the additional feature of being able to interact with the host language.

Obviously, the quantity of code necessary to write a specification in a black-box system is generally smaller than the C++ code necessary to instantiate EASY LOCAL++. Nevertheless, in our opinion, it is not necessarily true that writing a few lines of code in Localizer or SALSA is easier than writing a hundred lines of C++ code. For instance, using EASY LOCAL++ one can exploit broadly-available and well-established C++ libraries for data structures (e.g. STL [20] or LEDA [1]) and powerful graphical debugging tools, which need to be developed from scratch for black-box systems.

Current constraint programming languages are also tools for the solutions of combinatorial search problems. The difference is that their built-in solvers only run on constraints expressed in their specific constraint syntax. Therefore, they force the user to express the definition of the problem within the given language, whereas in our case the representation of the problem is completely left to the user.

On the other hand, there are many reliable software tools available, such as ILOG Solver [21], which provides the user with a large set of built-in facilities for programming the search algorithms, thus making this paradigm very attractive. Up to now, however, they mostly provide exhaustive search algorithms, rather than local search ones. The implementation of local-search techniques is also currently underway in the ILOG system [22] as a C++ library. From [22], though, it seems that this local-search library is not a framework like EASY LOCAL++ (with inverse control), but basically a toolkit that is invoked by the user.

Another software tool for programming local-search algorithms is Localizer++ [23], the evolution of Localizer. The system consists in a C++ library for local search that provides a set of both declarative abstractions to describe the neighborhood and high-level search constructs to specify local moves and meta-heuristics concisely. Even though it is basically a toolkit rather than a OO framework, Localizer++ exhibits some important OO features such as a customizable hierarchy of classes for expressing and incrementally maintaining constraints in a declarative fashion, and the possibility to define new search constructs.

Glass-box systems: OO frameworks

Moving to glass-box systems, a few OO frameworks for local-search problems have been already developed and are described in the literature, notably in [24–29].

The system HOTFRAME [29,28] is a C++ framework for local search. HOTFRAME is fully based on the use of templates and inheritance is only used in a secondary way. In HOTFRAME the type of neighborhood, the tabu mechanism and other features are supplied through template classes and values. This results in a very compositional architecture, given that every specific component can be plugged in by means of a template instantiation.

On the other hand, it does not exploit the power of virtual functions, which help in the development of the system and user modules. For example, in EASY LOCAL++ it is possible to change the behavior
of an algorithm at run-time, simply by replacing one of its components. Using templates, instead, this should be done at compile-time. In addition, in HOTFRAME several member functions need to be defined for the template instantiation classes. In EASYLOCAL++, conversely, such classes are simply data structures and the ‘active’ role is played exclusively by the helper classes. On the other hand, HOTFRAME comes with several predefined components for a whole bunch of popular problem representations, which are still under development for EASYLOCAL++.

Ferland and co-workers [25,26] propose an OO implementation of several local-search methods. In particular, in [25], they provide a framework developed in Object-Oriented Turbo Pascal. Differently from our work, their framework is restricted to assignment type problems only and, therefore, they are able to commit to a fixed structure for the data of the problem.

Specifically, our template class Move corresponds in their work to a pair of integer-valued parameters \((i, j)\), which refer to the index \(i\) of an item and the new resource \(j\) to which it is assigned, similarly to a finite-domain variable in constraint programming. Such a pair is simply passed to each function in the framework. Similarly, our template class State is directly implemented as an integer-valued array. The overall structure of the framework is, therefore, greatly simplified and most of the design issues related to the management of problem data do not arise. This simplification is obviously achieved at the expense of the generality and flexibility of the framework.

De Bruin et al. [24] developed a template-free framework for branch and bound search, which shows a different system architecture. Specifically, in their framework solver classes are concrete instead of being base classes for specific solvers. The data for the problem instance is supplied by a class, say MyProblem, derived from the framework’s abstract class Problem. The reason why we do not follow this idea is that the class MyProblem should not only contain the input and output data, but also all the functions necessary for running the solver like, for example, ComputeCost() and SelectMove(). Therefore, the module MyProblem would have less cohesion with respect to our solution that uses the modules Input, Output and the concrete solver class.

The description of other notable systems, such as Searcher [30] and HSF [31], can be found in a recent collection [4]. The collection also includes the presentation of systems that implement other optimization paradigms, such as constraint programming, and different types of meta-heuristics (e.g. genetic algorithms and scatter search). The collection provides a comprehensive picture of the state-of-the-art of optimization software libraries at the time of publication of this paper.

A further description of related work, including systems that implement other search techniques, like ABACUS [32] and KIDS [33], is provided in [34]. It also describes LOCAL++, the predecessor of EASYLOCAL++, which is composed of a single hierarchy of classes, without the distribution of responsibilities between helpers, runners and solvers.

LOCAL++ architecture showed several limitations that led to the development of EASYLOCAL++. For example, the code that in EASYLOCAL++ belongs to the helpers had to be duplicated for each technique in LOCAL++. In addition, LOCAL++ missed the ability to freely compose the features of the algorithms so as to give rise to a variety of new search strategies. Furthermore, LOCAL++ did not support many other important features of EASYLOCAL++, including the weight managing capabilities, the testers, the skeleton code and the experiment language EXPSpec.

Finally, as already mentioned, EASYLOCAL++ has been made freely available for the community and it has already been downloaded by many researchers. The continuous exposure to critics and comments by other researchers has given us additional motivations to extend and improve the system.
CONCLUSIONS

In this paper, we have presented EASY LOCAL++ an OO framework for the implementation of local-search algorithms. EASY LOCAL++ gives complete freedom to the user in terms of data structures and variable domains, but provides a fixed structure for controlling the execution flow.

The main goal of EASY LOCAL++ and similar systems is to simplify the task of researchers and application people who want to implement local search algorithms. Unfortunately, though, in many cases it is these problem-specific details that dominate the total implementation time for a local-search algorithm, so one might at first wonder why they should bother automating the ‘easy’ part.

The answer to these criticisms is twofold. First, recent research has proven that the solution of complex problems goes toward the direction of the simultaneous employment of various local-search techniques and neighborhood relations. Therefore, the ‘easy’ part tends to increase in complexity and programming cost. Second, we believe that EASY LOCAL++ provides the user with added value not only in terms of quantity of code, but also in modularization and conceptual clarity. Using EASY LOCAL++, or other OO frameworks, the user is forced to place each piece of code in the ‘right’ position.

EASY LOCAL++ makes a balanced use of the OO features required for the design of a framework. In fact, on the one hand, data classes are provided through templates, giving a better computational efficiency and a type-safe compilation. On the other hand, the algorithm’s structure is implemented through virtual functions, giving the chance of incremental specification in hierarchy levels and a complete reverse control communication. We believe that, for local search, this is a valid alternative to toolkit systems à la ILOG Solver.

One of the main characteristics of EASY LOCAL++ is its modularity: once the basic data structures and operations are defined and ‘plugged-in’, the system provides a straight implementation of all standard techniques and a large variety of their combinations for free.

The system also allows the user to generate and experiment with new combinations of features (e.g. neighborhood structures, initial state strategies and prohibition mechanisms) with a conceptually clear environment and a fast prototyping capability.

Moreover, we have exemplified the applicability of EASY LOCAL++ by the implementation of various algorithms for the graph coloring problem. We also discussed some of the strategic design decisions of the framework and their underlying motivations.

The current modules have actually been applied to a few practical problems.

- University examination timetabling: schedule the exam of a set of courses in a set of time-slots avoiding the overlapping of exams for students and satisfying other side constraints.
- Employee timetabling (or workforce scheduling): assign workers to shifts ensuring the necessary coverage for all tasks, respecting workload regulations for employees.
- Portfolio selection: select a portfolio of assets (and their quantity) that provides the investor with a given expected return and minimizes the associated risk. Differently from the other two, this problem makes use of both integer and real variables.

Several other modules are under implementation and testing. For example, we are working on a module that implements the Variable Neighborhood Search of Hansen and Mladenović [35], which makes use of a set of neighborhood relations of increasing size and performs a combination of our token-ring and comparative strategies. In addition, a threading mechanism is ongoing, which would...
manage the parallelization of the execution (see, e.g. [36,37]). Future work also comprises an adaptive tool for semi-automated framework instantiation in the style of the Active CookBooks proposed in [38], in order to help the users to develop their applications.

ACKNOWLEDGEMENTS

We would like to thank Marco Cadoli and two anonymous referees, whose advice helped us to improve this paper. This research was partially funded by Optiware Italia, Trieste, Italy, and University of Udine, Italy.

REFERENCES