A case-study for EasyLocal++: the Course Timetabling Problem

Luca Di Gaspero[1] and Andrea Schaerf[2]

[1] *Dipartimento di Matematica e Informatica*
Università degli Studi di Udine, via delle Scienze 206
I-33100 Udine, Italy
e-mail: `digasper@dimi.uniud.it`

[2] *Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica*
Università degli Studi di Udine, via delle Scienze 208
I-33100 Udine, Italy
e-mail: `schaerf@uniud.it`

Research Report UDMI/13/2001/RR

Abstract

EasyLocal++ is an object-oriented framework that helps the user to design and implement local search algorithms in C++ for a large variety of problems.

In this paper we highlight the usability of EasyLocal++ by showing its contribution for the development of a solver for a real-life scheduling problem, namely the CourseTimetabling problem.

The CourseTimetabling problem involves hard and soft constraints, and requires, in order to be solved in a satisfactory way, a non-trivial combination of different neighborhood relations. We show all steps of the implementation using EasyLocal++, which in our opinion is very straightforward. The resulting code is modular, small, and easy to maintain.

Introduction

Local search is a paradigm for optimization which is based on the idea of navigating the search space by iteratively stepping from one state to one of its "neighbors". The neighborhood of a state is composed by the states which are obtained by applying a simple local change to the current one.

Due to this simple schema, the developers of local search algorithms usually write their algorithms from scratch. By contrast, we believe that a clear conceptual methodology can help the user in the development of a local search application, in terms of both the quality of the software and its possible reuse. To support this claim, we design and developed EasyLocal++: an object-oriented (O-O) framework to be used as a general tool for the design and the implementation of local search algorithms in C++.

A framework is a special kind of O-O library, which consists of a network of abstract and concrete classes, that are used through inheritance. The idea is that the framework provides most of the high-level structures of the program, and the user must only define suitable derived classes and implement the virtual functions of the framework. Thanks to the use of virtual functions, frameworks are characterized by the so-called *inverse control* mechanism: The functions of the framework call the user-defined ones and not the other way round.

Therefore EasyLocal++, as a framework for local search, provides the full control of the invariant part of the local search algorithm, and the user classes only supply the problem-specific details and the neigborhood relations.

EasyLocal++ is described in details in [2]. In this paper, we recall its main features and show a comprehensive case study of its use for the solution of a complex scheduling problem, namely the so-called CourseTimetabling problem.

There are various formulations of the CourseTimetabling problem (see e.g., [7]), which differ from each other mostly for the hard and soft constraints (or objectives) they consider. For the sake of simplicity, we describe here a basic version of the problem. Nevertheless, we have also implemented, using EasyLocal++, a solver for the more complex version which applies to the Faculty of Engineering of the University of Udine. This latter version is actually in use to generate the real timetable of the Faculty.

In addition, we describe a new feature of EasyLocal++, not presented in [2], which allows the user to build composite neighborhoods, called *kickers*, starting from simple ones. As discussed in Section 4, kickers turned out to be very helpful for the solution of the CourseTimetabling problem.

Both EasyLocal++ and the CourseTimetabling solver described in this paper are available from the web page `http://www.diegm.uniud.it/schaerf/projects/local++`.

1 An Overview of EasyLocal++

Before describing the main features of EasyLocal++, we first briefly recall the local search paradigm. Unfortunately, though, there is not a full agreement in the research

```
procedure LocalSearch(Neighborhood N, cost function f);
begin
   s_0 := InitialSolution();
   i := 0;
   while (¬StopSearch(s_i, i)) do
   begin
      m := SelectMove(s_i, f, N);
      if (AcceptableMove(m, s_i, f))
      then   s_{i+1} := s_i ⊕ m;
      else   s_{i+1} := s_i;
      i := i + 1
   end
end;
```

Figure 1: The abstract local search procedure

community on what the term "local search" precisely includes. We discuss here how we intend it, and consequently the way it is implemented in EasyLocal++.

1.1 The Local Search Paradigm

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

Given an instance p of a problem P, we associate a *search space* S to 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 and enters a loop that *navigates* the search space, stepping from one state s_i to one of its neighbors s_{i+1}. The neighborhood is usually composed by the states that are obtained by some local changes (called *moves*) from the current one.

Local search metaheuristics are therefore built upon the abstract local search algorithm reported in Figure 1; they differ from each other according to the strategy they use to select the move in each state, to accept it, and to stop the search (encoded in the *SelectMove*, *AcceptMove* and *StopSearch* functions, respectively). In all techniques, the search is driven by a *cost function* f that estimates the quality of the state. For optimization problems, f generally accounts for the number of violated constraints and for the objective function of the problem.

The most common local search metaheuristics are *hill climbing* (HC), *simulated annealing* (SA), and *tabu search* (TS) (see, e.g., [1] for a detailed review).

One attractive property of the local search paradigm is that different metaheuristics

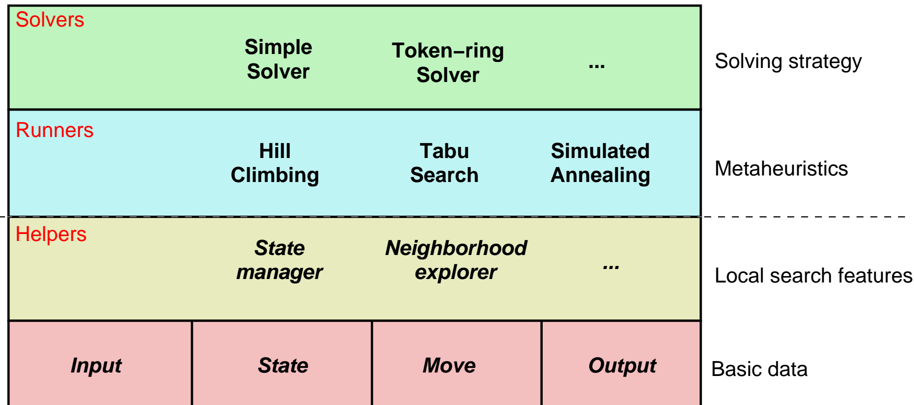

Figure 2: The levels of abstraction in EasyLocal++

can be combined and alternated to give rise to complex techniques. An example of a simple mechanism for combining different techniques and/or different neighborhood relations is what we call the *token-ring* strategy: Given an initial state and a set of basic local search techniques, the token-ring search makes circularly a run of each technique, always starting from the best solution found by the previous one. The search stops when no improvement is obtained by any algorithm in a fixed number of rounds.

The effectiveness of token-ring search for two runners, called *tandem* search, has been stressed by several authors (see [4]). In particular, when one of the two runners 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., [8]).

The ability to implement in an easy and clean way complex techniques is in our opinion one of the strengths of EasyLocal++.

1.2 The EasyLocal++ architecture

The core of EasyLocal++ is composed by a set of cooperating classes that take care of different aspects of local search. The user's application is obtained by writing derived classes for a selected subset of the framework ones. 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 invocation, are completely dealt with by the framework.

The classes of the framework are split in four categories, and are organized in a hierarchy of abstraction levels as shown in Figure 2. Each layer of the hierarchy relies on the services supplied by lower levels and provides a set of more abstract operations, as we are going to describe.

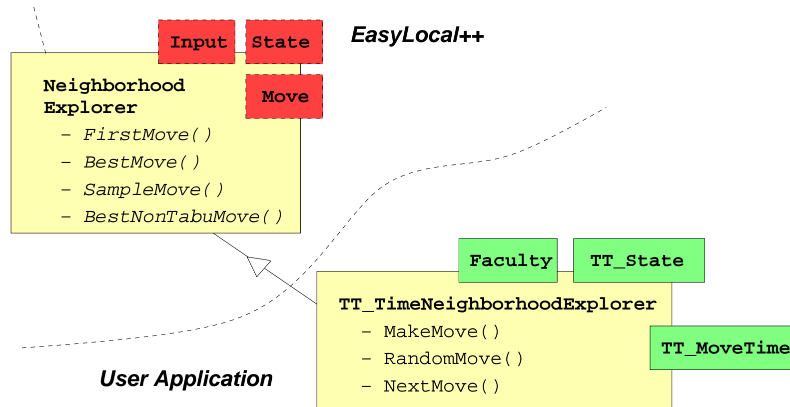

Figure 3: A step of the development process

1.2.1 Data Classes

Data classe are the lowest level of the stack. They maintain the problem representation (class `Input`), the solutions (class `Output`), the states of the search space (class `State`), and the attributes of the moves (class `Move`).

Data classes only store attributes, and have no computing capabilities. The data classes are supplied to the other classes as templates, which need to be instantiated by the user with the corresponding problem-specific types.

1.2.2 Helpers

The local search features are embodied in what we call *helpers*. These classes perform actions related to each specific aspect of the search. For example, the Neighborhood Explorer is responsible for everything concerning the neighborhood: selecting the move, updating the current state by executing a move, and so on. Different Neighborhood Explorers may be defined in 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 computation of the cost function, and delegates this task to the State Manager which handles the attributes of each state. Helpers do not have their own internal data, but they work on the internal state of the runners, and interact with them through function parameters.

Methods of EasyLocal++ helper classes are split in three categories that we call *MustDef*, *MayRedef*, *NoRedef* functions.

***MustDef*:** *pure virtual* C++ functions that correspond to problem specific aspects of the algorithm; they *must* be defined by the user.

***MayRedef*:** *non-pure virtual* C++ functions that come with a tentative definition, which *may* be redefined by the user in case the definition is not satisfactory for the problem

at hand. 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 and they *cannot* be redefined by the user.

1.2.3 Runners and Solvers

Runners represent the algorithmic core of the framework. They are responsible for performing a run of a local search algorithm, starting from an initial state and leading to a final one. Each runner has many data objects for representing the state of the search (current state, best state, current move, number of iterations, ...), and it maintains links to all the helpers, which are invoked for performing problem-related tasks on its own data.

Runners can completely abstract from the problem description, and they delegate these tasks to user-supplied classes which comply to a predefined interface.

This allows us to describe metaheuristics through incremental specification: For example, we can directly translate the abstract local search algorithm in C++ code in EasyLocal++. Then, we specify actual metaheuristics, by defining the strategy for move selection and acceptance (through the *SelectMove*() and *AcceptableMove*() functions, respectively), and the criterion for stopping the search (by means of the *StopSearch*() function). Three main metaheuristics have been implemented in EasyLocal++, namely Hill Climbing, Simulated Annealing and Tabu Search.

The highest abstraction level is constituted by the *solvers*, which represent the external software layer of EasyLocal++. Solvers control the search by generating the initial solutions, and deciding how, and in which sequence, runners have to be activated.. A solver, for instance, implements the token-ring strategy described before; other solvers implements other combinations of basic metaheuristics and/or hybrid methods.

Solvers are linked to the runners (one or more) that belong to their solution strategy. In addition, solvers communicate with the external environment, by getting the input and delivering the output.

For runners and solvers, all functions are either *MayRedef* or *NoRedef*, which means that their use requires only to define the appropriate derived class (see the case study below).

New runners and solvers can be added by the user as well. This way, EasyLocal++ supports also the design of new metaheuristics and the combination of already available algorithms by the user. In fact, it is possible to describe new abstract algorithms (in the sense that they are decoupled from the problem at hand) at the runner level, while, by defining new solvers, it is possible to prescribe strategies for composing pools of basic techniques.

1.2.4 Testers

In addition to the classes presented, the framework provides a set of *tester* classes, which act as a generic user interface of the program and support both interactive and batch runs of the algorithms. Batch runs can be programmed using a dedicated language, called

ExpSpec, which allows to compare different algorithms and parameter settings with very little intervention of the human operator.

In order to use the framework, the user has to define the data classes and the derived helper, runner, and solver classes which encode the specific problem description. An example of a step of this process is reported in Figure 3 in the case of an algorithm for the CourseTimetabling problem (introduced in Section 2). The function names drawn in the box `TT_TimeNeighborhoodExplorer` are *MustDef* functions that must be defined in the user's subclass, whereas the functions in the box `NeighborhoodExplorer` are *MayRedef* or *NoRedef* ones that are thus already defined in the framework. The data classes `Faculty`, `TT_State`, and `TT_MoveTime`, defined by the user, instantiate the templates `Input`, `State`, and `Move`, respectively.

1.2.5 Kickers

A new feature of EasyLocal++ (not present in [2]) is represented by the so-called *kickers*. A kicker is a complex neighborhoods composed by chains of moves belonging to base neighborhoods. The name "kicker" (due to [8] up to our knowledge) comes from the metaphor of a long move as a kick given the current state in order to perturb it.

Among others, a kicker allows the user to draw a random kick, or to search for the best kick of a given length. In principle, a kicker can generate and evaluate chains of arbitrary length. However, due to the size of the neighborhood, finding the best kick is generally computationally infeasible for lengths of 3 or more. To reduce the computational cost, the kickers can be programmed to explore only kicks composed by certain combinations of moves. In details, a kicker searches for a chain of moves that are "related" to each other. The intuitive reason is that kickers are invoked when the search is trapped in a deep local minimum, and it is quite unlikely that a chain of unrelated moves could be effective in such situations.

The notion of "related moves" is obviously problem dependent, therefore the kicker classes include one *MustDef* function called `RelatedMoves()` that takes as argoment two moves and return a boolean value. The user writes the complete definition for her/his specific problem. An example of a kicker, and various definitions of `RelatedMoves()` are provided in Section 3.4.

2 The CourseTimetabling Problem

The university CourseTimetabling problem consists in scheduling the lectures of a set of courses of a university in the given set of time periods that compose the week and using the available rooms.

There are q courses $C_1, \ldots, C_q$, p periods $1, \ldots, p$, and r rooms $R_1, \ldots, R_r$. Each course C_i consists of l_i lectures to be scheduled in distinct time periods, and it is taken by s_i students. Each room R_j has a capacity c_j, in terms of number of seats.

The output of the problem is an integer-valued $q \times p$ matrix T, such that $T_{ik} = j$ (with

$1 \leq j \leq r$) means that course C_i has a lecture in room R_j at period k, and $T_{ik} = 0$ means that course C_i has no class in period k. We search for the matrix T such that the following *hard* constraints are satisfied, and the violations of the *soft* ones are minimized:

Lectures (hard): The number of lectures of each course C_i must be exactly $_i$.

Room Occupation (hard): Two distinct lectures cannot take place in the same room in the same period.

Conflicts (hard): There are groups of courses that have the students in common, called *curricula*. Lectures of courses in the same curriculum must be all scheduled at different times. Similarly, lectures of courses taught by the same teacher must also be scheduled at different times.

We define a conflict matrix CM of size $q \times q$, such that $cm_{ij} = 1$ if C_i and C_j cannot be scheduled in the same period, $cm_{ij} = 0$ otherwise.

Availabilities (hard): Teachers might be not available for some periods. We define an availability matrix A of size $q \times p$, such that $a_{ik} = 1$ if lectures of course C_i can be scheduled at period k, $a_{ik} = 0$ otherwise.

Room Capacity (soft): The number of students that attend a course must be less or equal than the number of seats of all the rooms that host its lectures.

Minimum working days (soft): The set of periods p is split in d days of p/d periods each (assuming p divisible by d). Each period therefore belongs to a specific week day. The lectures of each course C_i must be spread into a minimum number of days m_i (with $m_i \leq k_i$ and $m_i \leq d$).

This concludes the (semi-formal) description of CourseTimetabling. It is easy to recognize that it is a hard problem. In fact, the underlying decision problem ("does a solution satisfying the hard constraints exist?") can be easily shown NP-complete through a reduction from the graph coloring problem.

3 Solving CourseTimetabling using EasyLocal++

We now show the solution of the Course Timetabling problem using EasyLocal++. We proceed in stages: We start from the data classes, then we show the helpers, the runners, and finally the solvers. The use of the testers is presented in the next section, which is devoted to the execution of the software.

For the sake of simplicity, the classes presented below are simplified with respect to the version used in the actual implementation. For example, the input and output operators (">>" and "<<") and a few auxiliary data and functions are omitted.

Nevertheless, the full software is available from the web page given in Section .

3.1 Data Classes

The input of the problem is represented by the class `Faculty` which stores all data about courses, rooms, periods, and contraints.

```
class Faculty
{public:
  Faculty();
  void Load(string instance); // reads an instance from file(s)

  unsigned Courses() const { return courses; }
  unsigned Rooms() const { return rooms; }
  unsigned Periods() const { return periods; }
  unsigned PeriodsPerDay() const { return periods_per_day; }
  unsigned Days() const { return periods/periods_per_day; }

  bool  Available(unsigned c, unsigned p) const
      { return availability[c][p]; } // availability matrix access
  bool Conflict(unsigned c1, unsigned c2) const
     { return conflict[c1][c2]; } // conflict matrix access
  const Course& CourseVector(int i) const { return course_vect[i]; }
  const Room& RoomVector(int i) const { return room_vect[i]; }
  const Period& PeriodVector(int i) const { return period_vect[i]; }
  // ... data and auxiliary functions (omitted for brevity)
};
```

The classes `Course`, `Period`, and `Room` store data on the corresponding entities (such as name, capacity, location, ...), and we omit their definitions here. The function `Load()`, which reads data from file(s), is a mandatory member of the input class, and EasyLocal++ relies on its presence.

Second, we show the output class, that stores a solution of the problem.

```
class Timetable
{public:
  Timetable(Faculty* f = NULL);
  SetInput(Faculty* f);
  unsigned operator()(unsigned i, unsigned j) const { return T[i][j]; }
  unsigned& operator()(unsigned i, unsigned j) { return T[i][j]; }
protected:
  virtual void Allocate();
  vector<vector<unsigned> > T;
  Faculty* fp;
};
```

The class stores only the matrix T itself, but no information about the input data, which are supplied through `fp`, a pointer to the `Faculty` class.

The constructor with one argument sets the value of the pointer `fp`, and allocates the matrix `T`; if no argument is supplied, the constructor leaves the object uninitialized. The function `SetInput()` initializes (or reinitializes) an already existing object according to the provided input. Finally, the operators "`()`" simply access the matrix `T`.

As an example of the code, we show the functions `SetInput()` and `Allocate()`.

```
void Timetable::SetInput(Faculty * f)
{ if (fp != f)
    { fp = f;
      Allocate();
    }
}

void Timetable::Allocate()
{ T.resize(fp->Courses(), vector<unsigned>(fp->Periods(),0)); }
```

The constructors and `SetInput()` are the mandatory members for a class that instantiates the `Output` template.

The `Input` and `Output` classes (shown above) describe the problem itself and are independent of the search technique. We now move to the classes that implement local search.

We start from the class that instantiates the `State` template. To this aim, we have first to define the search space: We decide to use the set of all possible $q \times p$ output matrices T, with the additional condition that the requirements and all availabilities are satisfied. Thus, for this problem the class `TT_State` that instantiates the template `State` is similar to the class `Timetable` that instantiates the output template. This is not always the case, because in general the search space can be an indirect representation of the output.

The difference stems from the fact that `TT_State` includes also redundant data structures used for efficiency purposes. These data store aggregate values, namely the number of lectures in a room per period, the number of lectures of a course per day, and the number of teaching days for a course.

Consequently, we define as the `State` template a class derived from `Timetable`, as shown below:

```
class TT_State : public Timetable
{public:
  TT_State(Faculty * f = NULL)  : Timetable(f) {}
  unsigned RoomLectures(unsigned i, unsigned j) const
          { return room_lectures[i][j]; }
  unsigned CourseDailyLectures(unsigned i, unsigned j) const
          { return course_daily_lectures[i][j]; }
  unsigned WorkingDays(unsigned i) const
          { return working_days[i]; }
```

```
  // ... (functions that manipulates redundant data omitted)
  protected:
   void Allocate();
   // ... (redundant data omitted)
};
```

Redundant data structures make the state updates more expensive, whereas they save time for evaluations of the state and its neighbors. Their presence is justified by the fact that, as we will see later, the number of evaluations of candidate moves (neighbors) is much larger than the number of state updates.

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()` (inherited in this case) are mandatory. Notice that in this case `SetInput()` is inherited, but it invokes the proper version of the `Allocate()` function (the one of the class `TT_State`), thanks to the late binding of virtual functions.

The fact that requirements and availabilities are always satisfied implies that they need not to be included in the cost function; on the other hand, functions that generate or modify states should be defined in such a way that these constraints are kept satisfied.

The first neighborhood relation that we consider is defined by the reschedule of a lecture in a different period (leaving the room unchanged). For implementing this kind of move, we define a class, called `TT_MoveTime`, as follows:

```
class TT_MoveTime
{public:
   unsigned course, from, to;
};
```

The effect of a move of this type is that the lecture of the course of index `course` is moved from the period `from` to the period `to`.

It is inportant to observe that not all possible values of a `TT_MoveTime` object represents a *feasible* move. A move is feasible if the value of `course` is included between 0 and $q-1$, and the values of `from` and `to` are between 0 and $p-1$ (remind that in C/C++ arrays start from 0); in addition, in the current state `st` there must be a lecture of course `course` in period `from` and none in period `to`, and the period `to` is available for the course.

The other move type that we consider, which is used in alternation to `TT_MoveTime`, is called `TT_MoveRoom` and its effect is to replace the room of a lecture of a given course. The corresponding data class is the following.

```
class TT_MoveRoom
{public:
   unsigned course, period, old_room, new_room;
};
```

A TT_MoveRoom move replaces the old_room with the new_room for the lecture of the given course scheduled in the given period. A move is feasible if course is between 0 and $q-1$, period is between 0 and $p-1$, and old_room and new_room are between 1 and r (0 represents the absence of a lecture). In addition, there must be a lecture of course in period in old_room.

Notice that in order to select and apply a TT_MoveRoom move from a given state st we only need the course, the period, and the new room. Nevertheless, it is necessary to store also the old room for the management of the prohibition mechanisms. In fact, the tabu list stores only the "raw" moves regardless of the states in which they have been applied.

3.2 Helpers

We need to define four helpers, namely the State Manager, the Output Producer, the Neighborhood Explorer, and the Prohibition Manager. Given that we deal with two move types, we have to define two Neighborhood Explorers and two Prohibition Managers, therefore we have to define six helpers all together.

We start describing the State Manager, which is represented by the following class.

```
class TT_StateManager : public StateManager<Faculty,TT_State>
{public:
  TT_StateManager(Faculty*);
  void RandomState(TT_State&);   // mustdef
 protected:
  fvalue Violations(const TT_State& as) const;   // mayredef
  fvalue Objective(const TT_State& as) const;    // mayredef

  void UpdateRedundantStateData(TT_State& as) const;  // mayredef
  void ResetState(TT_State& as);

  unsigned Conflitcs(const TT_State& as) const;
  unsigned RoomOccupation(const TT_State& as) const;
  unsigned RoomCapacity(const TT_State& as) const;
  unsigned MinWorkingDays(const TT_State& as) const;
   // ... other functions
};
```

We first describe the function RandomState() that assigns all lectures to random (but available) periods in random rooms.

```
void TT_StateManager::RandomState(TT_State& as)
{ ResetState(as); // make all elements of as equal to 0
  for (unsigned c = 0; c < p_in->Courses(); c++)
    { unsigned lectures = p_in->CourseVector(c).Lectures();
      for (unsigned j = 0; j < lectures; j++)
        { unsigned p;
```

```
          do // cycle until the period is free and available
            p = Random(0,p_in->Periods()-1);
          while (as(c,p) != 0 || !p_in->Available(c,p));
          as(c,p) = Random(1,p_in->Rooms());
        }
    }
  UpdateRedundantStateData(as);
}
```

The function is composed by a double cycle that assigns a (distinct) random period and a random room to each lecture. The data member `p_in` is a pointer to the input class, that is inherited from the abstract `StateManager`.

The functions `Violations()` and `Objective()` return the sum of the number of violations for each type of constraint (hard and soft)[1].

```
int TT_StateManager::Violations(const TT_State& as) const
{ return Conflitcs(as) + RoomOccupation(as); }

int TT_StateManager::Objective(const TT_State& as) const
{ return RoomCapacity(as) + MinWorkingDays(as); }
```

Among the functions that compose the cost, we show `RoomOccupation()` and `RoomCapacity()` omitting the other two that are similar.

```
unsigned TT_StateManager::RoomOccupation(const TT_State& as) const
{ unsigned cost = 0;
  for (unsigned p = 0; p < p_in->Periods(); p++)
    for (unsigned r = 1; r <= p_in->Rooms(); r++)
      if (as.RoomLectures(r,p) > 1)
        cost += as.RoomLectures(r,p) - 1;
  return cost;
}

unsigned TT_StateManager::RoomCapacity(const TT_State& as) const
{ unsigned cost = 0;
  for (unsigned c = 0; c < p_in->Courses(); c++)
    for (unsigned p = 0; p < p_in->Periods(); p++)
    { unsigned r = as(c,p);
      if (r != 0 &&
          p_in->RoomVector(r).Capacity() < p_in->CourseVector(c).Students())
        cost++;
    }
  return cost;
}
```

[1]Weights of soft constraints are assumed for simplicity to be all equal to 1.

Notice that the function `RoomOccupation()` uses the redundant structure `RoomLectures`. These structures are used more intensively by the functions of the Neighborhood Explorers that compute the variations of the cost (see below).

We now move to the description of the first Neighborhood Explorer, which is represented by the class `TT_TimeNeighborhoodExplorer`, defined as follows.

```
class TT_TimeNeighborhoodExplorer
  : public NeighborhoodExplorer<Faculty,TT_State,TT_MoveTime>
{public:
  TT_TimeNeighborhoodExplorer(StateManager<Faculty,TT_State>*, Faculty*);
  void RandomMove(const TT_State&, TT_MoveTime&);          // mustdef
  bool FeasibleMove(const TT_State&, const TT_MoveTime&);  // mayredef
  void MakeMove(TT_State&,const TT_MoveTime&);             // mustdef
 protected:
  fvalue DeltaViolations(const TT_State&, const TT_MoveTime&); // mayredef
  fvalue DeltaObjective(const TT_State&, const TT_MoveTime&);  // mayredef

  int DeltaConflitcs(const TT_State& as, const TT_MoveTime& mv) const;
  int DeltaRoomOccupation(const TT_State& as, const TT_MoveTime& mv) const;
  int DeltaMinWorkingDays(const TT_State& as, const TT_MoveTime& mv) const;
  void NextMove(const TT_State&,TT_MoveTime&);   // mustdef
 private:
  void AnyNextMove(const TT_State&,TT_MoveTime&);
  void AnyRandomMove(const TT_State&, TT_MoveTime&);
};
```

We present the implementation of some of the function of the class. We start with `NextMove()` that enumerates all possible moves and is needed to explore exhaustively the neighborhood.

```
void TT_TimeNeighborhoodExplorer::NextMove(const TT_State& as, TT_MoveTime& mv)
{ do
    AnyNextMove(as,mv);
  while (!FeasibleMove(as,mv));
}
```

For modularity, we separate the generation of moves in circular lexicographic order (function `AnyNextMove()`), from the feasibility check (function `FeasibleMove()`)

```
void TT_TimeNeighborhoodExplorer::AnyNextMove(const TT_State& as,
                                              TT_MoveTime& mv)
{ if (mv.to < p_in->Periods() - 1)
    mv.to++;
  else if (mv.from < p_in->Periods() - 1)
      { mv.from++;
```

```
        mv.to = 0;
      }
  else
    { mv.course = (mv.course + 1) % p_in->Courses();
      mv.from = 0;
      mv.to = 1;
    }
}

bool TT_TimeNeighborhoodExplorer::FeasibleMove(const TT_State& as,
                                               const TT_MoveTime& mv)
{  return as(mv.course,mv.from) != 0 && as(mv.course,mv.to) == 0
          && p_in->Available(mv.course,mv.to);
}
```

Next, we show the function `MakeMove()` that performs the move by changing the state and updating the redundant data accordingly.

```
void TT_TimeNeighborhoodExplorer::MakeMove(TT_State& as, const TT_MoveTime& mv)
{     // update the state matrix
  unsigned room = as(mv.course,mv.from);
  as(mv.course,mv.to) = room;
  as(mv.course,mv.from) = 0;

     // update the redundant data
  unsigned from_day = mv.from / p_in->PeriodsPerDay();
  unsigned to_day= mv.to / p_in->PeriodsPerDay();
  as.DecRoomLectures(room,mv.from);
  as.IncRoomLectures(room,mv.to);

  if (from_day != to_day)
    {
      as.DecCourseDailyLectures(mv.course,from_day);
      as.IncCourseDailyLectures(mv.course,to_day);
      if (as.CourseDailyLectures(mv.course,from_day) == 0)
        as.DecWorkingDays(mv.course);
      if (as.CourseDailyLectures(mv.course,to_day) == 1)
        as.IncWorkingDays(mv.course);
    }
}
```

Finally, we show two of the functions that compute the difference of violations that a move would cause, namely `DeltaRoomOccupation()` and `DeltaMinWorkingDays()`.

```
int TT_TimeNeighborhoodExplorer::DeltaRoomOccupation(const TT_State& as,
                                                     const TT_MoveTime& mv) const
```

```
{ int cost = 0;
  unsigned r = as(mv.course,mv.from);
  if (as.RoomLectures(r,mv.from) > 1) cost--;
  if (as.RoomLectures(r,mv.to) > 0) cost++;
  return cost;
}

int TT_TimeNeighborhoodExplorer::DeltaMinWorkingDays(const TT_State& as,
                                                    const TT_MoveTime& mv) const
{
  unsigned from_day = mv.from / p_in->PeriodsPerDay();
  unsigned to_day = mv.to / p_in->PeriodsPerDay();

  if (from_day == to_day) return 0;
  if (as.WorkingDays(mv.course) <= p_in->CourseVector(mv.course).MinWorkingDays()
      && as.CourseDailyLectures(mv.course,from_day) == 1
      && as.CourseDailyLectures(mv.course,to_day) >= 1)
    return 1;
  if (as.WorkingDays(mv.course) < p_in->CourseVector(mv.course).MinWorkingDays()
      && as.CourseDailyLectures(mv.course,from_day) > 1
      && as.CourseDailyLectures(mv.course,to_day) == 0)
    return -1;
  return 0;
}
```

Notice how the above functions do not compute the values in the two states, but focus exactly on the changes. This way, thanks also to the redundant data, they have constant computational cost instead of quadratic.

The Prohibition Manager associated with the class `TT_TimeNeighborhoodExplorer` is represented by the class `TT_TimeTabuListManager`. The full code of the class, which consists of a constructor and of the only function that needs to be defined, `Inverse()`, is included within the class definition as shown below.

```
class TT_TimeTabuListManager : public TabuListManager<TT_MoveTime>
{
public:
  TT_TimeTabuListManager(int min = 0, int max = 0)
    : TabuListManager<TT_MoveTime>(min,max) {}
protected:
  bool Inverse(const TT_MoveTime& m1, const TT_MoveTime& m2) const
  { return m1.course == m2.course && (m1.from == m2.to || m2.from == m1.to); }
};
```

Notice that according to the above definition of the function `Inverse()`, we consider a move inverse of another one if it involves the same course and it has one of the two periods on common.

The Neighborhood Explorer based on the TT_MoveRoom moves is implemented by the class TT_RoomNeighborhoodExplorer. This class is very similar to TT_TimeNeighborhoodExplorer, except that the functions that contribute to DeltaViolations() and DeltaObjective() are only DeltaSamePeriodSameRoom() and DeltaRoomCapacity(), respectively, because only these types of constraints are interested in TT_MoveRoom() moves.

We omit the definition of the class, but we show the three functions RandomMove(), MakeMove(), and DeltaRoomOccupation().

```
void TT_RoomNeighborhoodExplorer::RandomMove(const TT_State& as, TT_MoveRoom& mv)
{ mv.course = Random(0,p_in->Courses() - 1);
  do
    mv.period = Random(0,p_in->Periods() - 1);
  while (as(mv.course,mv.period) == 0);
  mv.old_room = as(mv.course,mv.period);
  do
    mv.new_room = Random(1,p_in->Rooms());
  while (mv.new_room == mv.old_room);
}

void TT_RoomNeighborhoodExplorer::MakeMove(TT_State& as, const TT_MoveRoom& mv)
{
  as(mv.course,mv.period) = mv.new_room;
  as.DecRoomLectures(mv.old_room,mv.period);
  as.IncRoomLectures(mv.new_room,mv.period);
}

int TT_RoomNeighborhoodExplorer::DeltaRoomOccupation(const TT_State& as,
                                                     const TT_MoveRoom& mv) const
{
  int cost = 0;
  if (as.RoomLectures(mv.old_room,mv.period) > 1)
    cost--;
  if (as.RoomLectures(mv.new_room,mv.period) > 0)
    cost++;
  return cost;
}
```

We omit the corresponding Prohibition Manager, which is straightforward.

The Output Producer transforms a state into a solution, and vice versa. It also reads and writes solutions from file, in order to store results of current runs, and use them as starting point for future runs. In our case, given that the Timetable and TT_State classes are very similar, its functions are very simple. We show for example the function InputState(), that passes from an output to a state.

```
void TT_OutputManager::InputState(TT_State& as, const Timetable& tt) const
{ for (unsigned i = 0; i < p_in->Courses(); i++)
    for (unsigned j = 0; j < p_in->Periods(); j++)
      as(i,j) = tt(i,j);
  p_sm->UpdateRedundantStateData(as);
}
```

3.3 Runners and Solvers

We create runners which implement HC and TS for each of the two move types. No function needs to be defined for these four runners, except the constructor (which in C++ is not inherited).

The definition of the HC runner for TT_MoveTime move is the following.

```
class TT_TimeHillClimbing : public HillClimbing<Faculty,TT_State,TT_MoveTime>
{public:
  TT_TimeHillClimbing(StateManager<Faculty,TT_State>* psm,
      NeighborhoodExplorer<Faculty,TT_State,TT_MoveTime>* pnhe, Faculty* pin)
        : HillClimbing<Faculty,TT_State,TT_MoveTime>(psm,pnhe,pin)
            { SetName("HC-Timetabler"); }

};
```

The constructor only invokes the constructor of the base EasyLocal++ class, and sets the name of the runner. The name is necessary only to use the runner inside the Tester as described in Section 4.

The definition of the other three runners is absolutely identical, and therefore it is omitted.

We define now the token-ring solver, which is used for running various tandems of runners. The runners participating to the solver are selected at run-time by using the functions AddRunner() and ClearRunners(); therefore, the composition does not require any other programming effort, but, similarly to the runners, the solvers' derivation is only a template instantiation.

```
class TT_TokenRingSolver : public TokenRingSolver<Faculty,Timetable,TT_State>
{public:
  TT_TokenRingSolver(StateManager<Faculty,TT_State>* psm,
        OutputManager<Faculty,Timetable,TT_State>* pom,
        Faculty* pin, Timetable* pout)
  :  TokenRingSolver<Faculty,Timetable,TT_State>(psm,pom,pin,pout) {}
};
```

This completes the description of the software belonging to the solver itself. If the solver is embedded in a larger project, the extra code necessary to run it obviously depends on

the host program. If instead the solver is stand-alone, we resort to the tester, a module of EasyLocal++ which allows the user to run (and debug) the solver, as explained in Section 4.

We describe separately the kickers, which in the current version of EasyLocal++ cannot be invoked by the solver, but interact only with the tester.

3.4 Kickers

In the current version of EasyLocal++, a kicker can be either *monomodal* or *bimodal*, which means that is has one or two atomic neighborhood components, respectively.

For our problem, we have seen that we have two move types, therefore we define a bimodal kicker that deals with them. The definition of the class is the following.

```
class TT_TimeRoomKicker : public BimodalKicker<Faculty,TT_State,TT_MoveTime,
                                               TT_MoveRoom>
{
public:
  TT_TimeRoomKicker(TT_TimeNeighborhoodExplorer *tnhe,
                    TT_RoomNeighborhoodExplorer *rnhe);
  bool RelatedMoves(const TT_MoveTime &mv1, const TT_MoveTime &mv2);
  bool RelatedMoves(const TT_MoveTime &mv1, const TT_MoveRoom &mv2);
  bool RelatedMoves(const TT_MoveRoom &mv1, const TT_MoveTime &mv2);
  bool RelatedMoves(const TT_MoveRoom &mv1, const TT_MoveRoom &mv2);
};
```

As already mentioned, the four functions `RelatedMoves()` are used to limit the possible compositions of the kicks (chains). They are four different definitions, rather than a single more complex one because of different parameter types. As an example, we provide two of the four definitions:

```
bool TT_TimeRoomKicker::RelatedMoves(const TT_MoveTime &mv1,
                                     const TT_MoveRoom &mv2)
{ return mv1.to == mv2.period; }

bool TT_TimeRoomKicker::RelatedMoves(const TT_MoveRoom &mv1,
                                     const TT_MoveRoom &mv2)
{ return mv1.period == mv2.period && mv1.new_room == mv2.old_room; }
```

The first one states that a `TT_MoveRoom` move `mv2`, in order to be related to the `TT_MoveTime` move `mv1` that precedes it, must regard the period in which `mv1` has rescheduled the lecture (i.e., `mv1.to`). Similarly, a `TT_MoveRoom` move `mv2` is related to another `TT_MoveRoom` move `mv1` if they regard the same period and `mv2` removes a lecture in the room occupied by `mv1`.

These definitions are justified intuitive by the observation that chain of moves that are not related, in the above sense, rarely give a joint improvement to the cost function.

We have concluded the code needed in order to use the kicker. In fact, the search for the best kick is completely done by an exhaustive search implemented abstractly in EasyLocal++, by the function `BestKick()`. It relies on the functions of the helper, like `NextMove()`, and on a vector of states that stores the intermediate states generated by the kick move.

The experiments show that the kicker takes a few minutes to find the best kick of size 3. Although this is a undoutbly a long time, quite surprisingly, a kick of size 3 is often able to improve on the best solutions found by the token-ring solver.

4 Debugging and Running the Solver

The current version of EasyLocal++ includes a text-based tester, which acts as a generic user interface. In the future, a graphical interface will be hopefully available.

The tester stores pointers to all the objects involved in the solution (helpers, runners, solvers, and kickers) and activates them accurding to user's inputs.

The tester can run in two modes: interactive or batch. In the interactive mode, it maintains (and displays) a current state and proposes to the user a set of possible actions upon the current state. The actions are grouped in three menus:

Move Menu: Allows the user to choose one neighborhood relation (both simple or composite), and to select and perform one move belonging to that neighborhood. The move can be either the best one, or a random one, or supplied by the user. The state obtained by the execution of the move becomes the new current one. The user can also request additional information or statistics about neighborhoods, such as their cardinality, the percentage of improving moves, and so on.

Run Menu: Allow the user to select one runner, input its parameters, and run it. The best state reached by the runner becomes the new current state of the tester.

State menu: This menu includes all utilities necessary for managing the current state: The user can write it to a file, replace it with a new one read from file or generated at random, print all violations associated with it, and so on.

To be precise about the State menu, what the tester writes to the file is not the state itself, but rather the solution (output). This is because, in general, the state, which can be an implicit representation of the solution, doesn't necessarily have an intuitive meaning. To this regard, the tester uses the Output Producer, which tranlates states into solutions and vice versa. A fragment of an example of a (random) output file is given in Fig. 4, where `27`, `GRT`, `DIS`, etc. are room names.

The output is stored in this form, which is both human- and computer-readable, so that the user can interact with the tester. For example, he/she can write the current state on a file, modify it by hand, have the tester read it again, and invoke a runner on it.

```
Course  (Teacher) |       Monday        |       Tuesday        |
 - - - - - - - - -|  9   11   14   16  |  9   11   14   16  |  9   ...
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ArcTec  (Rossi)   |  -    -    -   GRT |  E    -   27    -  |  D   ...
TecCos  (Gialli)  |  -    -   36    -  | 30    -    -    -  |  DIS ...
GeoTec  (Verdi)   |  -    -   30    -  |  -   30    -    -  |  29  ...
CosIdr  (Neri)    |  -    -   30    -  |  -    -   31    F  |   -  ...
CalStr  (Viola)   |  -    -    -    -  |  -    -    A    -  |  DIS ...
IngTer  (Rossi)   |  -    -    -    -  |  -    -   27    -  |   -  ...
IdrApp  (Marroni) |  -    -    -    -  |  -   37    D    -  |   A  ...
RilAmb  (Bianchi) |  L    -    -    -  |  -    -    A    -  |   -  ...
EcoOrg  (Verdi)   |  -    -    E    -  |  -    -    -    -  |   -  ...
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 4: The appareance of the output timetable

In the batch mode, the tester reads a file written in the language ExpSpec (see [2]), and performs all experiments specified in the file. An example of a fragment of a ExpSpec file is the following.

```
Instance "FirstTerm"
{ Trials: 20;
  Output prefix: "TS2";
  Runner tabu search "TS-Timetabler"
  { min tabu tenure: 25;
    max tabu tenure: 40;
    max idle iteration: 1000;
    max iteration: 20000;
  }
  Runner tabu search "TS-Roomtabler"
  { min tabu tenure: 10;
    max tabu tenure: 10;
    max idle iteration: 500;
  }
}
```

The above ExpSpec block instructs the tester to run 20 trials of the token-ring solver composed by the two TS runners on the instance called `FirstTerm`.

Various parameters are specified for each runner. For example, let's analyze the parameters of the first runner: The fact that `min tabu tenure` and `max tabu tenure` are different prescribes that the tabu list must be dynamic. Specifically, this means that when a move enters in the list at iteration i, it is assigned a random number between $i + 25$ and $i + 40$. This number represents the interation in which the move will leave the tabu list. In this way, the size of the list is not fixed, but varies dynamically in the interval 25–40.

The stop criterium is twofold: the search is interrupted either after 1000 iteration without an inprovement of the best state, or after 20000 total iterations.

According to the specification, the best states of the trials are written into a set of files whose names start with `TS2` (i.e., `TS2-01.out`, `TS2-02.out`, ..., `TS2-20.out`). At the end of a block of experiments like this, the tester gathers statistics about results and running times, and then it moves to the next specification block.

The batch mode is especially suitable for massive night or weekend runs, in which the tester can perform all kinds of experiments in a completely unsupervised mode.

5 Discussion and Conclusions

The main goal of EasyLocal++, and similar systems, is to simplify the task of researchers and practitioners who want to implement local search algorithms. The idea is to leave only the problem specific programming details to the user. We have presented the main part of this process for a non-trivial scheduling problem.

The architecture of EasyLocal++ prescribes a precise methodology for the design of a local search algorithm. The user is required to identify exactly the entities of the problem at hand, which are factorized in groups of related classes in the framework: Using EasyLocal++ the user is forced to place each piece of code in the "right" position. We believe that this feature helps in term of conceptual clarity, and it makes easier the reuse of the software and the overall design process.

Furthermore, we have shown that the composition of the basic entities in EasyLocal++ is straightforward. The user can obtain with a limited effort a set of local search algorithms, possibly based on different neighborhood relations, which can be composed in different ways.

A few other systems for local search or other techniques have been proposed in the literature, such as Localizer++ [6], HotFrame [3], and ABACUS [5]. A comparison between EasyLocal++ and those systems is provided in [2].

Here we just want to mention that, in our opinion, the strength of EasyLocal++ is that it makes a balanced use of O-O features needed for the design of a framework, namely templates and virtual functions. 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, algorithm's structure is implemented through virtual functions, giving the chance of incremental specification in hierarchy levels and providing the inverse control communication.

This balance does not fit perfectly for every component. For example, there is an implementation problem about kickers. Recall that we have defined monomodal and bimodal kickers, however it would be very handful to define a generic *multimodal* kicker, which would be composed by a dynamic number of neighborhood relations. Unfortunately, due to the fact that in EasyLocal++ moves are supplied through templates (see [2] for the reasons of this design choice), it is not possible to define such a multimodal kicker without static type-checking violations.

References

[1] Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Son, Chichester, 1997.

[2] Luca Di Gaspero and Andrea Schaerf. EasyLocal++: An object-oriented framework for flexible design of local search algorithms. Technical Report UDMI/13/2000/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2000. Available at `http://www.diegm.uniud.it/schaerf/projects/local++`.

[3] Andreas Fink, Stefan Voß, and David L. Woodruff. Building reusable software components for heuristc search, 1998. Extended abstract of the talk given at OR98, Zürich.

[4] Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[5] Michael Jünger and Stefan Thienel. The design of the branch-and-cut system ABACUS. Technical Report TR97.263, University of Cologne, Dept. of Computer Science, 1997.

[6] Michel Laurent and Pascal Van Hentenrick. Localizer++: An open library for local search. Technical Report CS-01-02, Brown University, 2001.

[7] Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[8] Thomas Stützle. Applying iterated local search to the permutation flow shop problem. Technical Report AIDA-98-04, FG Intellektik, TU Darmstadt, 1998.