CORE
Metadata, citation and similar papers at core.ac.uk
Provided by UPCommons. Portal del coneixement obert de la UPC

Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Generic Parallel Implementations for Tabu Search

Maria J. Blesa, Jordi Petit, Fatos Xhafa

Report LSI–05–50–R

16th November 2005

Generic Parallel Implementations for Tabu Search*

Maria Blesa† Jordi Petit Fatos Xhafa

ALBCOM Research group
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Ω building Campus Nord, E-08034 Barcelona, Spain
{mjblesa,jpetit,fatos}@lsi.upc.edu

Abstract

Tabu Search (TS) is a meta-heuristic for solving combinatorial optimization problems. A review of existing implementations for TS reveals that, on the one hand, these implementations are *ad hoc* and, on the other hand, most of them run in a sequential setting. Indeed, the reported parallel implementations are few as compared to the sequential implementations. Due to increase in computing resources, especially in LAN environments, it is quite desirable to obtain parallel implementations of TS for solving problems arising in fields others than computer science, such as biology, control theory, etc., in which researchers and practitioners are less familiar with parallel programming.

In this work we present a generic implementation of TS able to be run in sequential and parallel settings. The key point in our approach is the design and implementation in C++ of an *algorithmic skeleton* for TS embedding its main flow as well as several parallel implementations for the method. This is achieved through a separation of concerns: elements related to TS are provided by the skeleton, whereas the problem-dependent elements are expected to be provided by the user according to a fixed interface using purely sequential constructs. Thus, the skeleton has a unique interface but is expected to have many instantiations for concrete problems, all of them being able to run in a straightforward way using different parallel implementations.

In order to assess the effectiveness of our approach, we have applied it to several NP-hard combinatorial optimization problems. We have considered developing time, flexibility and easiness of use, quality of solutions and computation efficiency. We have observed that our approach allows fast developing of problem instantiations. Moreover, the skeleton allows the user to configure and implement in different ways internal methods related to TS. Furthermore, the results obtained by our generic parallel implementations are efficient and report good quality results compared to the ones reported by *ad hoc* implementations.

We exemplify our approach through the application to the 0–1 Multidimensional Knapsack problem. The experimental results obtained for standard benchmarks of this problem show that, in spite of the genericity and flexibility of our implementation, the resulting program provides high quality solutions very close to the optimal ones.

Keywords. Combinatorial Optimization, Algorithmic skeletons, Tabu Search, $0-1$ Multidimensional Knapsack.

*Work partially supported by the Spanish MCYT project TIC2002-04498-C05-02 (TRACER). At the time while research was conducted, M. Blesa was also supported by the Catalan Research Council of the Generalitat de Catalunya under grant no. 2001FI-00659. Preliminary versions of some parts of this work were presented at EUROPAR'02 ([1], Section 4), ICPADS'01 and PPAM'01 ([4, 5], Sections 4 and 5). Those versions included introductory ideas and experiments on some of the models presented here. This paper extends the contents of those papers both in proposing new models, and in including more experimental results. Moreover, this work unifies all the set of generic parallel models thus providing a complete survey/overview for the reader.

†Corresponding author

1 Introduction and motivation

Many interesting combinatorial optimization problems are known to be NP-hard [19] and hence unlikely to be solvable within a reasonable amount of time. Heuristic methods have proved to be a good alternative to cope in practice with such problems [7]. One such method is Tabu Search (TS) introduced by F. Glover [21, 22].

TS has been applied to many combinatorial optimization problems such as scheduling [26, 41, 14, 32], graph problems [24, 6], resource allocations [35, 37], and layout problems [36, 17, 25] among others. After a careful revision of such implementations, we have observed that researchers and practitioners have applied TS to their problems through *ad hoc* implementations. This approach has, at least, two drawbacks. First, one has to implement the meta-heuristic from scratch for any new problem of interest and, second, even small changes in the code are not trivially introduced since this would require the modification of different parts of the implementation.

These drawbacks are even more patent when trying to obtain parallel implementations. Some researchers have investigated how to parallelize TS. In one hand, parallelism permits practitioners to use more resources to obtain better solutions in reasonable computing time, and in the other, the experience shows that parallel versions of heuristics tend to yield more robust implementations than the sequential ones, in the sense that they perform well in most instances of the problem [12]. A taxonomy of parallel TS strategies is presented in [12] and, for meta-heuristics in a general context, fundamental ideas to design parallel strategies are found in [11]. Those ideas are applicable to a wide range of problems, yet the existing parallel implementations of TS for different problems [38, 16, 31] remain *ad hoc*. So, while it is almost clear how to exploit parallelism in TS in a generic way, practitioners tend to re-implement from the scratch the whole heuristic and its parallelization using specific knowledge of the problem at hand.

The Generic Programming (GP) paradigm is a way to cope with these limitations. GP has turned out to be very useful to parallel programming [15, 13, 34, 20, 39, 23] in a manner that it allows good expressibility, reuse, and robustness yet maintaining efficiency. Combinatorial optimization is an area of applicability of this paradigm. Indeed, in combinatorial optimization we are often encountered with algorithms for sub-optimally solving a large number of optimization problems which apply in a similar manner to most of the problems they aim to solve. For instance, the main flow of TS applies similarly to any problem, and therefore it is quite interesting to have a *generic program* or a *skeleton* for TS from which one could derive instantiations for any problem of interest. Moreover, it would be interesting to endow such a skeleton with capabilities to run in parallel setting.

Furthermore, after years of research and development, parallel programming remains a difficult and specialized task. Yet, due to the increase in computing resources, especially in LAN environments, it is quite desirable to obtain reusable parallel implementations of meta-heuristics in general, and TS in particular, for solving problems arising in fields others than computer science, where researchers and practitioners are less familiar with parallel programming.

Our main objective in this work is to present a generic implementation of TS that allows in a easy and flexible way to obtain sequential and parallel programs for different problems. To this end, in this paper we present the design, the implementation and the evaluation of an *algorithmic skeleton* for TS. Our skeleton is the result of an abstraction process leading to the separation of two concerns: elements proper to TS are provided by the skeleton, whereas the problem-dependent elements are expected to be provided by the user. The link between them is achieved through a fixed interface. The main flow of TS and different ways to parallelize it are implemented by the skeleton since, in a generic way, they are proper to the method. On the other hand, problem-dependent elements are specified by the skeleton. Their interface, as well as the user's own instantiation of them, are described using purely sequential constructs. Thus, the skeleton has a unique interface but is expected to have many instantiations for concrete problems. Moreover, several parallel implementations are provided by the skeleton. We consider here the following parallel implementations for the TS skeleton: Independent Runs, Independent Runs with Autonomous Search Strategies, Master-Slave and Master-Slave with Neighborhood Partition.

In order to assess the effectiveness of our approach, we have applied it to several NP-hard com-

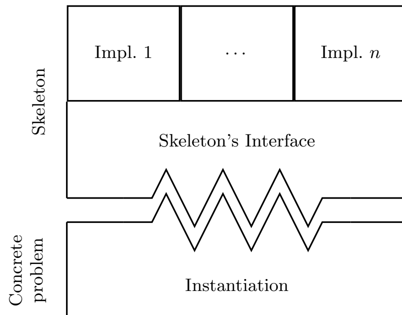


Figure 1: Structure of an skeleton

binatorial optimization problems: 0–1 Multidimensional Knapsack, $k-$Cardinality Tree, Max SAT, Max CUT, Resource Constrained Scheduling, Quadratic Assignment and Traveling Salesman. In this paper, we exemplify the approach through the application to 0–1 Multidimensional Knapsack problem. We consider developing time, flexibility and easiness of use, quality of solutions and computation efficiency.

The paper is organized as follows. In Section 2 we give some preliminaries on basic concepts from combinatorial optimization and in Section 3 we overview the TS meta-heuristic. The TS skeleton is presented in Section 4 and the parallel models used in the skeleton are given in Section 5. The application of our approach to the 0–1 Multidimensional Knapsack problem is shown in Section 6. We end the paper in Section 7 with some conclusions.

2 Preliminaries on combinatorial optimization problems

To describe more in detail how TS works, we need to introduce some definitions and notations concerning combinatorial optimization problems in general.

A combinatorial optimization problem is defined as follows.

Definition 1. A combinatorial optimization problem Π is a four-tuple $(\mathcal{I}, \mathcal{S}, c, goal)$ where:

- $\mathcal{I}$ is the set of instances,
- $\mathcal{S}(x)$ is the set of feasible solutions associated to any instance $x \in \mathcal{I}$,
- c is a cost function that maps feasible solutions $s \in \mathcal{S}(x)$ of a given instance x to values $c_x(s)$, referred to as the cost of the solution,
- a goal: find a feasible solution that optimizes (maximizes, minimizes) the cost function.

Given an instance x of a problem Π, the set $\mathcal{S}(x)$ of all feasible solutions is called the solution space. Solving a combinatorial optimization problem means to find a solution $s^* \in \mathcal{S}(x)$ such that s^* has the best cost with respect to the cost function, i.e., $c_x(s^*) \leq c_x(s)$, for all s in $\mathcal{S}(x)$, in the minimization case and $c_x(s^*) \geq c_x(s)$, for all s in $\mathcal{S}(x)$, in the maximization case. Such a solution is called optimal for the given instance x of the problem Π. Hereafter, to simplify notation, we will refer only to the minimization case and denote by $\mathcal{S}$ the set of feasible solutions $\mathcal{S}(x)$.

For many important combinatorial optimization problems, finding the optimal solution is computationally hard since the solution space of such problems is very big (grows exponentially in instance size). However, in many fields sub-optimal solutions, i.e. solutions whose cost is close to the optimum, might be satisfactory. Sub-optimal solutions are usually the best among a subset of feasible solutions, and are referred to as locally optimal solutions, in contrast to optimal solutions that are called globally optimal solutions. A simple way to visualize the property of locally versus globally optimal solutions is through the graphical representation of the landscape of the problem (see Fig. 2). In such a graphical representation we can see different valleys corresponding to locally minimal solutions and at least one of them could also be a globally minimal solution.

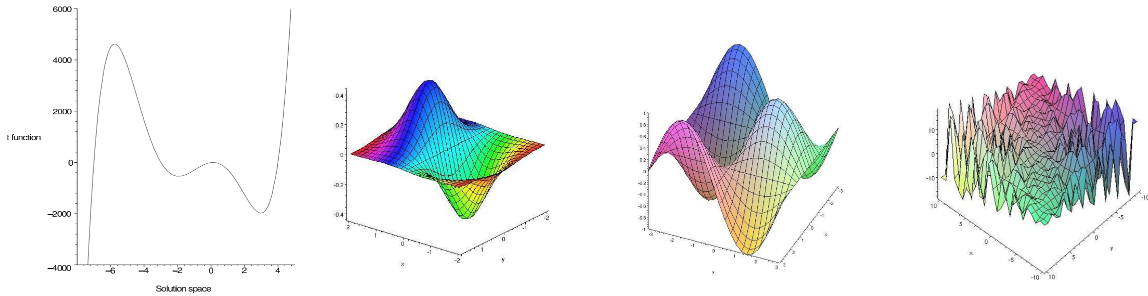


Figure 2: Different landscapes of continuous functions

In order to formally define locally minimal solutions, a neighborhood structure is introduced, which allows to view locally minimal solutions as those which fulfill the minimization criterion at least with respect to their neighborhood.

Definition 2. A *neighborhood structure* is a function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ that assigns to every solution $s \in \mathcal{S}$ a set of solutions $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is called the neighborhood of s and the solutions $s' \in \mathcal{N}(s)$ are called *neighbors* of s.

Then, a locally minimal solution is defined as follows.

Definition 3. A solution $\hat{s} \in \mathcal{S}$ is a *local minimum* with respect to a neighborhood structure $\mathcal{N}$ if the cost function is minimized for $\hat{s}$, i.e., if for all s in $\mathcal{N}(\hat{s})$ it holds that $c(\hat{s}) \leq c(s)$.

Note that for a given problem many different neighborhood structures can be defined. Usually, the definition of the neighborhood takes into account the combinatorial structure of solutions. In this sense, we can see the neighbors of a solution as those solutions that differ slightly from the given solution. These small changes are usually known as local perturbations and are interpreted as *movements*.

Definition 4. A *movement* $m : \mathcal{S} \to \mathcal{S}$ is a mapping from one solution $s \in \mathcal{S}$ to another solution $s' \in \mathcal{S}$.

Note that having the definition of a movement m we can define a neighborhood structure as follows: $\mathcal{N}(s) = \{s' \mid s' \in \mathcal{S},\ m(s) = s'\}$.

Local search methods use movements to explore the solution space by iteratively jumping from a feasible solution to another one. Depending on how the neighborhood is defined and which additional strategies are used to guide the search, different families of local search methods can be defined. For an overview of local search methods see [29].

3 The Tabu Search meta-heuristic

Heuristic methods have turned out to be a standard approach in combinatorial optimization. Dealing in practice with real size problems makes the use of such methods the *de facto* choice.

Tabu Search (TS) was formally introduced by F. Glover [21, 22]. As it happens with other heuristics such as Simulated Annealing, Genetic and Evolutionary Algorithms, TS has a clear link with real life experiences and behaviors. Indeed, TS adopts the traditional usage of the *"tabu"* word meaning prohibition imposed by social custom as a protective measure. This adoption has the following senses: (a) Some actions must be tabu since allowing them might be risky; (b) Some actions might be allowed in spite of being tabu; (c) Tabu actions are memorized, i.e., the search should keep track of the trajectories followed in the solution space and learn from the experience so that the search could be effective; and (d) The search should also benefit from exploring new promising regions of solutions space. TS method tries to incorporate all these observations through different mechanisms in a way that allows an effective search that together with its flexibility can beat many classical memoryless methods and obtain a remarkable efficiency on several problems.

High level description. TS starts from an initial solution and proceeds iteratively from a solution s to another one $s' \in \mathcal{N}^*(s)$, where $\mathcal{N}^*(s)$ denotes a reduced neighborhood, $\mathcal{N}^*(s) \subseteq \mathcal{N}(s)$. The search proceeds until a termination condition is met.

There are different ways to define the reduced neighborhood $\mathcal{N}^*(s)$, for instance, by labelling the recently visited solutions as tabu and storing them in a tabu list L. This constitutes the short term memory or recency. Then, in a given iteration, $\mathcal{N}^*(s)$ is computed as $\mathcal{N}(s)\backslash L$. Note that by forbidding solutions that are already visited, $\mathcal{N}^*(s)$ contributes to avoid cycling, though there is no guarantee that cycling will not occur. In order to maintain the tabu list in a reasonable size, the tabu status of solutions in L is canceled after a certain number of iterations.

It is important to note here that although we speak of a list of tabu solutions, a tabu solution refers to the tabu movement that leads to that solution. The tabu status of a movement is canceled also in exceptional conditions, known as aspiration criteria.

Based on its historical memory, the algorithm may decide to activate appropriately two mechanisms: intensification and diversification. The intensification is intended to explore more thoroughly the immediate neighborhoods of the current solution. It may happen, however, that during the search the algorithm gets stuck into a local optimum while there might still be interesting regions of solution space to be explored. In such a case the algorithm activates the diversification mechanism. See Fig. 3 for an intuitive idea of the effect of these two mechanisms.

We proceed now to explain in more detail the main entities used in the TS meta-heuristic and then give its main flow.

The initial solution. This is the starting point of the search and is usually constructed either at random or by a greedy procedure.

The neighborhood/movement. The definition of the reduced neighborhood $\mathcal{N}^*(s)$ of a solution s is crucial. Such a definition is done through the entity movement that is usually defined and implemented in terms of attributes of a solution. TS has to define and manage efficiently the movement entity and functionalities related to it such as "apply a movement to a solution", "compute the inverse movement", "give tabu status to a movement", etc. Note that there can be multiple types of movements whose application leads to neighborhoods dynamically changing and covering different regions of the solution space.

Memory types and tabu criteria. TS maintains historical information of the exploration process through a short term memory and a long term memory.

– A short term memory or recency is used to maintain information on recently visited solutions, usually by storing values of pre-selected attributes of such solutions and labelling them as tabu. One way of doing this is by considering as pre-selected attributes those forming the movement, hence, such a movement is labelled tabu and cannot be applied, unless it satisfies certain conditions. Note, however, that a movement might include attributes that are not pre-selected, so it will be labelled tabu if it contains some tabu attribute. Using the tabu attributes and tabu movements the algorithm reduces the neighborhood: solutions that contain tabu attributes or combinations of them in form of movements do not belong to $\mathcal{N}^*(s)$.
– A long term memory or frequency is used to maintain historical information on the exploration process. The objective now is to identify elements that are common to good solutions. This is done either by storing attributes appearing frequently in good solutions or by recording complete solutions (elite solutions), say, solutions whose cost is below a threshold. The long term memory is used to activate the intensification and diversification strategies.

The intensification. The purpose of this procedure is to explore in depth certain regions of the solution space if there were evidence that such regions may contain good solutions. To this end, the procedure tries to insert good attributes to the newly generated solutions. This is done by

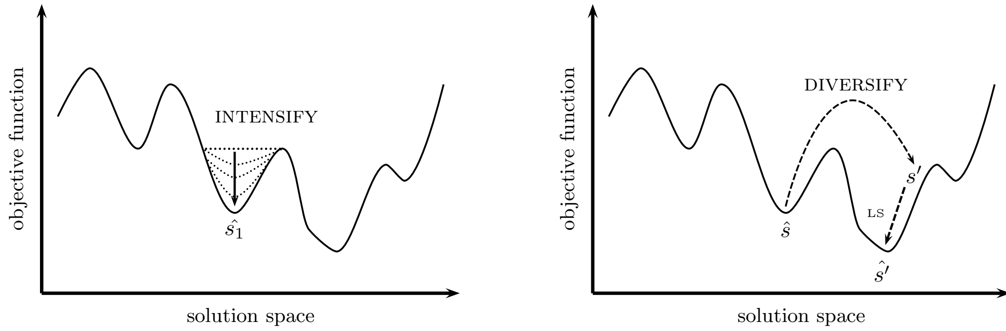


Figure 3: Intensification (left) and diversification (right) effects.

generating movements with attributes from those recorded in the frequency or from elite solutions and applying them through the standard movement mechanism. Another way to accomplish this is through rewarding attributes of the solution that are in the frequency and thus "forcing" these attributes to appear in the newly generated solutions.

The diversification. This procedure is intended as a mechanism to escape from local optima and to ensure that the algorithm will explore new regions of the solution space. Again, this is done using the frequency but now the attributes *not* frequently used are chosen. As in the case of intensification, it can also be done by penalizing attributes of the solution that are in the frequency and thus "forcing" these attributes to not appear in the newly generated solutions. We refer to this kind of diversification as "soft" diversification since it allows to move the search to neighborhoods close to the current one. Another (more abrupt) way to diversify is to re-launch the search at a new point in the solution space; we refer to this kind of diversification as "hard" diversification.

The aspiration criteria. This is a set of conditions to verify whether the tabu status of a movement can be canceled allowing thus its application. One such example is the improve-best aspiration criteria: if applying the tabu movement to the current solution leads to a better solution than the best one so far, then the tabu status of the movement is canceled.

The stopping condition. At each iteration, the stopping condition is evaluated. Usually, the stopping condition is a compound of different conditions: some of them related to TS, such as "during the exploration process, an empty $\mathcal{N}^*(s)$ is found", some others are related to pragmatical issues such as "the total number of iterations or the execution time is completed."

We conclude this section presenting the main procedure of TS in Algorithm 1. We can easily observe that the main procedure yields to the concept of the generic algorithm since it applies independently of the problem being solved; knowledge specific to the problem is needed only in implementing methods and entities used as black boxes in the main procedure.

4 The Tabu Search Skeleton

We present now the TS skeleton for the TS meta-heuristic described in the previous section.

4.1 Design of skeletons for generic algorithms

Our design for the TS skeleton follows the MALLBA approach described in [1]. This approach has been successfully used in the design of several other generic algorithms for combinatorial optimization such as Divide and Conquer, Dynamic Programming, Simulated Annealing, Memetic Algorithms and others [27].

Algorithm 1 : Pseudo code for Tabu Search

```
Input: A problem instance and setup parameters
Output: A sub-optimal solution ŝ
Compute an initial solution s ∈ S and ŝ ← s.
Reset the tabu and aspiration conditions.
while not stopping condition do
  Generate a subset N*(s) ⊆ N(s) of solutions such that, either none of the tabu conditions is
  violated or the aspiration criteria holds.
  Choose the best s' ∈ N*(s) with respect to the cost function c.
  s ← s'.
  if c(s') < c(ŝ) then
    ŝ ← s'.
  end if
  Update the recency and frequency.
  if intensification condition then
    Perform intensification procedure.
  end if
  if diversification condition then
    Perform diversification procedure.
  end if
end while
Return ŝ.
```

Algorithmic skeletons were formally introduced by M. Cole [9]. His idea was to capture common patterns of parallel computation, such as divide and conquer or pipelining, in higher-order functions. These patterns can then be instantiated by the programmer to suit a particular algorithm. The approach has the advantage of restricting parallelism to a small, easily isolated part of the program. The basic idea behind this is to not require the user to know any issue regarding parallelism. Rather, it is the implementation written by the skeleton developer that is aware of the possible ways to parallelize the execution. In fact, the same skeleton can be used for different architectures: it is only necessary to change the implementation of the skeleton in order to change the program's behavior, its instantiation itself is unchanged.

In the MALLBA approach, skeletons are targeted towards combinatorial optimization methods rather than towards basic parallel patterns. Also, rather than using functional languages, the skeleton's interface and its instantiation are described with purely sequential C++ constructs. The parallel constructs are hidden in the different implementations of the skeleton. This organization allows the user to instantiate any problem of his choice by only defining the concrete problem-dependent features that depend on the problem he has in mind using a widely available object oriented language. Abstract elements related to the inner algorithmic functionality of the method and its parallelization are hidden to him.

In order to design a skeleton for a given generic algorithm, one must first identify its basic abstract entities and functionalities. Then, one must describe their corresponding abstractions into C++ elements: classes and functions. These elements can be classified according to their "*availability*": Elements implementing inner functionalities of the generic algorithm (e.g. its main flow) are completely provided by the skeleton, whereas elements whose behavior is problem specific are required to be implemented by the user. Therefore, we classify these elements into two groups: *provided* and *required*.

- *Provided elements:* Provided classes and functions implement the generic algorithm itself. These may include the solvers that fix the main flow of execution as well as helper classes needed to get the state of the process or to setup some of its parameters.
- *Required elements:* Required classes and functions represent the entities and functionalities involved in the generic algorithm whose implementation depends on the specific problem

being solved. Their requirements are defined by the interfaces of C++ classes with which the provided classes can interact.

It is clear that the provided elements of the skeleton are written by the skeleton's developer whereas the required elements must be completed by the user of the skeleton according to the specified interface.

The C++ classes of the skeletons are separated in three parts: (1) the signature (interface) of the classes and functions; (2) the implementation of the provided classes and, (3) the implementation of the required classes. In terms of files, the signatures of the classes and functions are located in the `skeleton.hh` file, the implementation of the provided elements is located in the `skeleton.pro.cc` file and the implementation of the required elements must be completed in the `skeleton.req.cc` file. The user is expected to describe its particular problem-dependent elements in the `skeleton.hh` and `skeleton.req.cc` files. All the elements defined and implemented in these files are grouped under a unique namespace.

4.2 The skeleton's interface for Tabu Search

Let us focus now on the actual design of our TS skeleton. In the following, we present in more detail how the TS entities and concepts have been translated into classes and functions. We concentrate now on its abstract behavior, deferring to the following section the matters related to its implementation.

It must be noted that in order to abstract the entities and functionalities of this meta-heuristic and to enable our skeleton to be as much generic as possible, we undertook a careful review of several different *ad hoc* implementations available in the literature [26, 41, 14, 32, 24, 6, 35, 37, 36, 17, 25].

The provided `Solver` class. The provided `Solver` class represents the main procedure of our skeleton as it directs its main flow of execution (see Algorithm 1), keeps track of all the internal features related to the search (such as intensification and diversification) and collects information about its state. The flow of execution is divided into three levels: the global level, the independent-run level, and the phase level:

– The global level is composed of several independent runs.
– An independent run corresponds to an execution of TS.
– A phase is an atomic transition from a current solution s to another solution s' that becomes the next point to continue the search. Note that s' can either be the best solution in the neighborhood of s, either the solution obtained after intensification or the one obtained after diversification (depending on the conditions that trigger these procedures). In the former case a phase would coincide with a search iteration, but in the latter cases it would be composed of some search iterations.

These three levels are useful for two purposes. In one hand, they enable the implementation of different parallel models: The global and the independent-run levels are oriented towards coarse grain parallelizations and the phase level is oriented towards fine grain parallelizations. On the other hand, these levels also enable the possibility to interact with other skeletons. This is useful when considering hybrid heuristics [2].

The state of the exploration basically consists of information about the best solution found so far and the values of those attributes describing the current point of the search. These values are maintained for the three execution levels.

The actual implementation of the behavior defined in this class is really implemented in its subclasses, one for each different implementation (see Fig .4). Currently, we provide a sequential implementation and four parallel different ones (Independent Runs, Independent Runs with Autonomous Strategies, Master Slave and Master Slave with Neighborhood Partition, see next section).

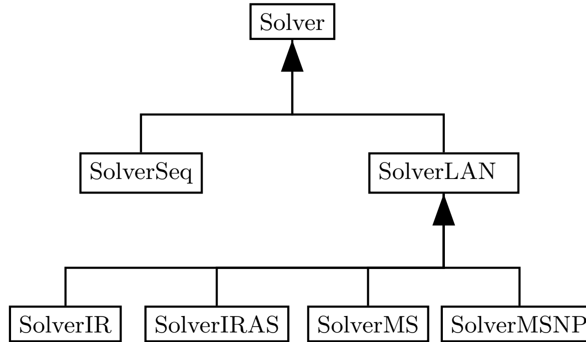


Figure 4: TS solvers.

All the methods in `Solver` are virtual and abstract. Despite this fact, as our skeleton is usually used, this virtual mechanism does not require any dynamic binding that could penalize execution speed. We also remark that the `provides` keyword is just an empty macro to help clarify the interface. Fig. 5 presents the complete interface of the `Solver` class.

The required `Problem` class. The required `Problem` class represents an instance of the problem to be solved. The internal implementation of the class only needs operations to create, serialize and get the direction of optimization (minimization or maximization). Their implementation will be provided by the user for a concrete problem. The interface of this class is shown in Fig. 6.

In passing, we remark that we use the `<<` and `>>` operators to provide standard input and output between files as well as to enable sending and receiving data between processes in the parallel implementations. This is accomplished in an independent way of the communication library by a communication module for which the user is not aware. We also remark that the `requires` keyword is just another empty macro to enhance the interface.

The required `Solution` class. The required `Solution` class represents a feasible solution for a given problem. Its interface is given in Fig. 7.

This class requires a method to compute the cost of a solution. However, it is often much more efficient to compute incrementally the cost of a new solution obtained by a movement from another solution than recomputing it from scratch. This is why we also include the possibility to use the `delta()` method, which returns the difference between the cost of the current solution and the cost of the solution obtained by applying a movement to it.

The generation of an initial solution must be implemented in the `set_initial()` method. The `perturb_randomly()` method is expected to randomly and locally change this solution, so that different searches can start at different points.

The `aspiration()` method is required to check the aspiration criteria so that tabu moves can be accepted. Intensification and diversification of solutions is done using these methods: `reward()` describes how the current solution (actually the items forming the solution) are rewarded, `unreward()` describes how to change back the solution to its original weight, `penalize()` and `unpenalize()` allow soft diversification in the search, and `escape()` performs hard diversification (see previous section).

The required `Movement` class. The interface of the required `Movement` class is given in Fig .8. The comparison operators are needed because of the management of the tabu movements. This management also requires the movement to have a “tabu life” (i.e. the time since a movement has been considered tabu), which can be set or retrieved. This class must also implement the `invert()` method to obtain the inverse movement of a movement.

```
provides class Solver {
public:
    Solver (const Problem& pbm, const Setup& setup);
    virtual ~Solver ();
    const Problem& problem () const;
    const Setup&   setup () const;

    // Execution and hybridization
    virtual void run () =0;
    virtual void perform_one_independent_run () =0;
    virtual void perform_one_phase () =0;
    virtual void set_current_solution (const Solution& sol);
    virtual void set_current_solution (const Solution& sol, const double cost);

    // Global information — state
    virtual int       independent_run () const;
    virtual double    time_spent () const;
    virtual Solution  best_solution () const;
    virtual int       independent_run_best_found () const;
    virtual int       iteration_best_found () const;
    virtual double    time_best_found () const;
    virtual double    best_cost () const;
    virtual double    worst_cost () const;

    // Independent run information — state
    virtual Solution  current_solution () const;
    virtual double    current_cost () const;
    virtual int       current_iteration () const;
    virtual Solution  current_initial_solution () const;
    virtual double    current_initial_cost () const;
    virtual double    current_time_spent () const;
    virtual Solution  current_best_solution () const;
    virtual int       current_best_solution_iteration () const;
    virtual double    current_best_solution_time () const;
    virtual double    current_best_cost () const;
    virtual double    current_worst_cost () const;

    // Phase information
    virtual int  nb_iterations_performed_in_phase () const;
    virtual bool intensification_in_phase () const;
    virtual bool soft_diversification_in_phase () const;
    virtual bool hard_diversification_in_phase () const;
};
```

Figure 5: The required `Solver` class.

```
enum Direction {maximization,minimization};

requires class Problem {
public:
    Problem ();
    ~Problem();
    friend ostream& operator<< (ostream& os, const Problem& pbm);
    friend istream& operator>> (istream& is, Problem& pbm);
    Direction direction () const;
};
```

Figure 6: The required `Problem` class.

```
requires class Solution {
public:

    Solution (const Problem&  pbm);
    Solution (const Solution& sol);
    ~Solution();

    Solution& operator=  (const Solution& sol);
    SolutionComponent& operator[] (int i);
    friend bool operator== (const Solution& sol1, const Solution& sol2);
    friend bool operator!= (const Solution& sol1, const Solution& sol2);

    friend ostream& operator<< (ostream& os, const Solution& sol);
    friend istream& operator>> (istream& is, Solution& sol);

    double cost () const;
    double delta (const Movement& move) const;
    int size () const;

    void set_initial ();
    void perturb_randomly ();

    void apply (const Movement& move );
    void unapply (const Movement& move );

    bool aspiration (const Movement& move, const TabuStorage& tstore, const Solver& solver) const;

    void reward ();
    void unreward ();
    void penalize ();
    void unpenalize ();
    void escape ();
};
```

Figure 7: The required `Solution` class.

```
requires class Movement {
public:
    Movement (const Problem& pbm, const Solution& sol);
    Movement (const Movement& move);
    Movement ();
    ~Movement ();

    Movement& operator= (const Movement& move);
    friend bool operator== (const Movement& move1, const Movement& move2);
    friend bool operator!= (const Movement& move1, const Movement& move2);

    friend ostream& operator<< (ostream& os, const Movement& move);
    friend istream& operator>> (istream& is, Movement& move);

    int  tabulife () const;
    void set_tabulife (int i);
    void invert ();
};
```

Figure 8: The required `Movement` class.

```
requires class TabuStorage {
public:
    TabuStorage (const Solver&);
    ~TabuStorage ();
    bool is_in_tabu_storage (const Movement& move, const Solution& sol) const;
    bool is_tabu (const Movement& move, const Solution& sol) const;
    void make_tabu (Movement& move, const Solution& sol);
    void make_tabu_inv (Movement& move, const Solution& sol);
    void update ();
    int  size () const;
};
```

Figure 9: The required `TabuStorage` class.

The required `TabuStorage` class. The `TabuStorage` class represents the structure to store and manage tabu movements and thus represents the memory of the search. Its interface is presented in Fig. 9. Its constructor needs a reference to the `Solver` object as a parameter because of the setup parameters. In particular, `tabu_size`, `max_tabu_status` and `min_tabu_status` influence its definition and management. Moreover, the solver also keeps information about the state of the exploration that can be useful to decide considering a movement tabu or not.

This class contains methods to make tabu a movement or the inverse of a movement. It also contains two methods to query the storage: `is_in_tabu_storage()` asks whether a movement is physically in the tabu storage, and `is_tabu()` asks whether a movement is or not tabu. Note that being considered tabu may be different than physically being in the tabu storage.

The `update()` method is applied from time to time to purge movements considered tabu for a long period of time (this may be configured using the `max_tabu_status` parameter in the setup). Finally, the `size()` method returns the size of the structure.

The required function to explore the neighborhood. The neighborhood of the current solution is explored by applying movements to it. The next solution in the exploration process will be the one obtained by applying the "best" movement to the current solution. The user decides what does "best" mean according to his criteria. Usually this decision is made according to the cost values and aspiration criteria of the solutions in the neighborhood. The user also decides whether the whole neighborhood or only a part of it will be explored. In order to specify how the neighborhood is explored, we require the `choose_best_move()` and `choose_best_move_from_partiton()`, `generate_strategy()`, functions whose interfaces are shown in Fig. 10.

To permit the user implement whatever concrete exploration method he wishes, this function receives the current `Problem`, `Solver` and `TabuStorage` objects. This function should return a boolean indicating if a movement was chosen and, if so, that movement is returned as an output parameter.

The required function to terminate the search. An exploration may finish when some termination criteria have been reached. In order to allow the user to specify any termination criteria he wishes in problem-independent way, we require him to complete the `terminate()` function. As the termination condition can depend on the setup parameters and the current state of the search, this decision is based on the current `Solver` object.

5 Parallel implementations in the TS skeleton

We have chosen four alternatives to parallelize TS and have implemented them into our skeleton. Two of them are based on coarse grain parallelism and the other two on fine grain parallelism. These parallel implementations are provided in the skeleton through subclasses of the `Solver_Lan` class and use the other required methods in the skeleton as we show below; see Fig, 4.

```
requires bool choose_best_move
            (const Problem& pbm, const Solution& sol,
             const TabuStorage& tstore, const Solver& solver,
             Movement& move);

requires bool choose_best_move_from_partition
            (const Problem& pbm, const Solution& sol,
             const array<SolutionComponent> partition,
             const TabuStorage& tstore, const Solver& solver,
             Movement& move);

requires bool terminate (const Solver& solver);

requires void generate_strategy (Setup& setup);
```

Figure 10: The required functions.

Independent Runs model. The Independent Runs model (IR) consists of simultaneous and independent executions of the same program. In this model, there is a processor doing the coordination task that consists in, at the beginning, sending the problem instance as well as the values for the parameters to the rest of processors and receiving the results upon termination of all the processors execution. At the end, the coordinator processor computes the best solution and may show other relevant statistics. In this model, each processor runs the same instance of the program on the same input data and the communication time is almost irrelevant. Observe that this model makes sense as far as the program is non-deterministic, which is precisely the case of meta-heuristic implementations that take random decisions. Note that running the same implementation in different processors usually leads to exploring different areas of the solution space via different search paths.

In general, running the parallel IR implementation on p processors is essentially equivalent to running the program p times sequentially since the overhead due to the parallelism (distributing the input and recollecting the results) is very small.

This model is accomplished in the skeleton through the `perform_one_independent_run()` method defined in the `Solver` class. Each slave processor executes, upon receiving the problem instance and the same setup parameters, the `perform_one_independent_run()` method through an instance of the `Solver_Seq` subclass provided in the skeleton. Note that in this implementation we exploit a coarse grain parallelism.

Independent Runs with Autonomous Strategies. The Independent Runs with Autonomous Strategies model (IRAS) is a generalization of the IR. In the IRAS model, a processor is given, additionally, a strategy to be used for its own search. A strategy consists of an initial solution and values for parameters that control the algorithm. Now, the coordinator processor, at the beginning, sends to any processor a strategy and the problem instance and receives the results upon termination of all the processors execution. Again, at the end, the coordinator processor computes the best solution, the best corresponding strategy and may show other relevant statistics.

The implementation in the skeleton is done through the `perform_one_independent_run()` method defined in the `Solver` class and the `generate_strategy()` method required by the skeleton.

Master Slave model. In the Master Slave model (MS) there are two distinguished types of processors: a *master* processor and processors called *slaves*. The control is performed by the master and the slaves are subordinated to it. The master processor spawns slaves processors, initializes them, assigns subtasks and collects their results. Then, it computes a result from the results obtained by the slaves and uses it for its own work and so on.

In our case, the master processor runs the TS algorithm and uses slaves to choose the best movement that leads to the best solution in the neighborhood of the current one. To this end,

each slave processor explores the neighborhood by its own and comes up with its best movement. Clearly, the task of exploring the neighborhood in parallel makes sense as far as the neighborhood exploration is not deterministic.

This model is achieved in the skeleton through the `perform_one_phase()` method defined in the `Solver` class and the `chose_best_move()` method required in the skeleton. The master processor runs the main flow of TS through the `perform_one_phase()` method and each slave processor executes, upon receiving the current solution (actually the current movement to be applied to the solution), the `chose_best_move()` method. Note that in this implementation we exploit a fine grain parallelism.

Master Slave with Neighborhood Partition. The Master Slave with Neighborhood Partition model (MSNP) is derived from the MS model by specifying the type of the task accomplished by the slave processors. In contrast to the MS, in this model each processor explores just a portion of the neighborhood. Thus, through this model, we can reduce the time needed to perform a complete neighborhood exploration. Note that in this case the neighborhood exploration can be deterministic.

This model is managed in the skeleton through the `perform_one_phase()` method defined in the `Solver` class and `choose_best_move_from_partition()` method required in the skeleton. The master processor runs the main flow of TS through the `perform_one_phase()` method and each slave processor executes, upon receiving the current solution (actually the current movement to be applied to the solution) and its partition, the `choose_best_move_from_partition()` method. As for the partition the master processor uses the `size()` and `operator[]` methods defined in the `Solution` class in order to partition the solution into as many (roughly equally) parts S_i as slave processors there are, and sends them to the slave processors. Then, the i-th slave processor uses S_i to explore the portion of the neighborhood through movements constructed with attributes from S_i and S according to the implementation of the `choose_best_move_from_partition()` method.

Other implementation issues. Our design has paid special attention to the independence of the implementation from the chosen communication library as well as the efficiency of the implementation. To do so, we have implemented high level primitives for sending/receiving that are used as black boxes by the skeleton. Thus, the skeleton implementation will not be affected if we use, say, PVM instead of MPI except for we have to implement the primitives for the desired communication library. The current implementation is done using MPI.

Regarding the efficiency of the implementations, we have tried to minimize the communication time, especially in the MS models. More precisely, in the implementation of the IRAS model, instead of using the `generate_strategy()` method by the coordinator processor to generate and send strategies to the slaves, each slave runs the method in order to generate its own strategy. This is possible due to the non-deterministic implementation of the method. Indeed, the initial solution is computed by `set_initial()` method of the class solution and then is perturbed using the `perturb_randomly()` method while the rest of tabu search parameters are randomly chosen in intervals of values specified by parameters given by the user. In the case of MS implementations, the efficiency of communications between the master processor and the slaves is even more crucial since the task performed by the slaves, that is, the neighborhood exploration is very frequent. To reduce the communication time, the master processor sends to the slave processors the initial solution and for the rest of phases just sends them the best resulting movement. This yields in increase of efficiency since solutions can be huge structures (e.g., a tree of thousands of edges) while movements contain few information (e.g., two edges to be swapped).

6 Case example: the 0–1 Multidimensional Knapsack

We have applied our approach to several combinatorial optimization problems as to assess its effectiveness. We exemplify the approach through the application to the 0–1 Multidimensional

Knapsack (0–1MKNP) through which we also show that our approach allows a fast developing as well as transparent access for the user to parallel executions. We present some experimental results both on sequential and parallel settings.

6.1 Problem statement

The 0–1MKNP problem consists in selecting a subset of n given items in such a way that the total profit of the selected items is maximized while knapsack constraints are satisfied. More formally, the problem can be stated as

$$\begin{array}{rll} \max & \sum_{j=1}^{n} c_j \cdot x_j & \\ \text{subject to} & \sum_{j=1}^{n} A_{i,j} \cdot x_j \leq b_i, & i = 1, \ldots, m \\ & x_j \in \{0,1\}, & j = 1, \ldots, n \end{array} \tag{1}$$

where $c_j \in \mathbb{N}$, $A_{i,j} \in \mathbb{N}$ and $b_j \in \mathbb{N}$. Each c_j is the profit associated to the item j. The components x_j are decision variables: $x_j = 1$ if the item j is selected and $x_j = 0$ otherwise. Each of the m knapsack constraints $\sum_{j=1}^{n} A_{i,j} \cdot x_j \leq b_i, i = 1, \ldots, m$, specifies a dimension of the knapsack. The special case of the 0–1MKNP with $m = 1$ is the classical knapsack problem.

6.2 Instantiating the TS skeleton for the 0–1 MKNP problem

In this section, we exemplify how the TS skeleton can be instantiated for solving the 0–1MKNP problem. We will highlight the main steps in the instantiation of the TS skeleton from the point of view of a final user.

As introduced in Sec. 4, when instantiating the TS skeleton for a concrete problem the user has to complete the interface (in the `.hh` file) by defining the data types that implements the main entities represented in the required classes. Then, methods of the required classes must be implemented (in the `.req.cc` file) according to the chosen representation. We explain the main data structures and some of the most relevant methods of each required class in the proposed instantiation. See Appendix A for more details.

Instantiating the `Problem` required class. We choose a representation for the benefits, for the capacities and for the constraint matrix. The former are implemented by two arrays of integers, while the latter is implemented by a two-dimensional array. Since the 0–1MKNP is a maximization problem, the `direction()` method returns `maximization`.

Instantiating the `Solution` required class. A feasible solution is represented by an array `contents` of binary values (the type of the solution components), in which the i-th position indicates if the i-th item is included in the knapsack or not (what the `operator[]` returns).[1] An initial feasible solution can be constructed in many different ways; for instance, in a greedy way by trying to insert into the knapsack as many items as possible with as greater benefits as possible.

The cost associated to a solution is the sum of the benefits of the items included in the knapsack. This evaluation will be done many times during the search process; thus, it is desirable to have a faster way of evaluating a solution. Since every intermediate solution in the search process is obtained by the application of a movement to the previous solution, the corresponding cost can also be computed incrementally more efficiently by considering how the cost variates when applying the movement. The `delta()` method does this.

The `apply()` method is implemented by marking in the movement which items have to be dropped from the knapsack, and those to be added. On the other hand, the `unapply()` method implements how the previous solution can be recovered from the current one.

[1] Moreover, every solution keeps a reference to the `Problem` class, because information concerning the constraints, capacities and benefits of the items is needed.

The best solutions found during the search process are used as a long-term memory and kept in the class `TabuStorage`. The items of the best solutions are used in the intensification and diversification processes (see Sec. 3). By rewarding an item j, its original benefit c_j is increased and the item will be more attractive to be kept as part of future solutions. In this way, the item is less susceptible to be chosen as a candidate to be dropped from the knapsack when moving through the neighborhoods. Similarly, we can reinforce this effect by also making less attractive (decreasing the benefit) those items which were rarely belonging to good solutions.

Soft diversification is done by studying the history of the items in the last explored solutions via the information in the `history` array (in the `TabuStorage` class) and the `history_rep` setup parameter. With this information, the `rep` value is calculated and used as a threshold for deciding whether an item has been very frequently used or not. Those items which are in the current solution and belonged more than `rep` times to the last visited solutions, are dropped from the current solution. Furthermore, those items which appeared in the last visited solution less than `rep` times, are incorporated to the current solution (if no capacity constraint is violated). The soft diversification process is implemented in the `penalize()` method. After `nb_diversifications` iterations, the soft diversification finishes. Note that the item benefits are not modified, so the `unpenalize()` method does not need implementation.

The strong diversification is implemented in the `escape()` method. A new initial solution is constructed similarly as in the `initial_solution()` method but randomly deciding, for each item, whether it is included or not in the knapsack.[2] The search process is re-launched from this new starting point.

Instantiating the `Movement` required class. A movement is defined as a set of items that are dropped from the current contents of the knapsack, together with a set of new items to be added. The items to be dropped are selected according to the saturation of the different dimensions of the knapsack. The items with the lowest benefit of the most saturated dimensions are candidates to be dropped. A candidate should not be tabu, because intuitively that would mean that it was inserted recently. The items to be added are selected according to the benefit they would produce. Those with greater benefit and not belonging to the current knapsack configuration are candidates to be added. The capacities of the knapsack must also be controlled and updated accordingly. Such a movement can be represented with an array of integers (each integer is the index of an item in the knapsack) and two integer values indicating the number of items to be dropped and the number of items to be added.

To construct this type of movements and their further management, we use the methods in the class and, for the sake of efficiency and modularity of the instantiation, we introduce two additional ones: the `can_apply()` method and the `belong()` method. Note that this is possible in our skeleton since these methods are used only in this required class and the rest of the classes do not need to be aware of them. The `can_apply()` method, which keeps track of the fact that a given item has to be added/dropped in the movement, is implemented by always filling the array first with the items that have to be dropped (at the `nb_drop` lowest positions of the array), and then the items that have to be added (at the `nb_add` last positions of the array). On the other side, the `belong()` method checks if a given item has already been considered in the movement. In this instantiation, the `generate()` and `invert()` methods do not need to be implemented, since the movement is generated item by item through consecutive applications of the `can_apply()` method.

Instantiating the `TabuStorage` required class. In order to deal efficiently with the historic information of the search process, we use the following data structures:

– A tabu list `tl` implemented as an array of size n. The j-th component stores the number of iteration where the j-th item is made tabu. This is used in the whole search process since it represents the short-term memory.

[2]This indeterminism reduces significantly the probability that the new search path starts with the same initial solution as the one produced by the greedy procedure implemented in method `initial_solution()`.

– A history array of size n in which the j-th component stores the number of iterations that the j-th item has been included in the knapsack. This is used in the soft diversification.
– A best_solutions array maintaining the nb_best_sols solutions of best benefits found so far. This information is used in the intensification.

This additional treatment of the history of the search process is aimed to provide good intensification and diversification effects, but it requires some extra parameters in the TabuStorage class. An example is the parameter nb_best_sols mentioned above, which specifies the maximum number of best solutions to store and becomes the dimension of the array best_solutions. Another parameter, namely history_rep, is included to decide whether an item has been incorporated many times or not into the solutions during the search. In addition to these parameters, other internal structures are needed to store and compute secondary data (e.g., item_frequencies).

The management of the tabu movements is simple. When a movement is made tabu, each of its items is included in the tabu list with a "time-stamp" recording the last time that this item was considered (added or dropped). The information in history also has to be properly updated any time a movement is made tabu. The update_best_sols() internal method checks whether the solution obtained by applying the movement that is actually made tabu, has to be stored in best_solutions. A movement is tabu (see is_tabu() method) if any of its items have been in the tabu list during more than tabu_size iterations. The items in the movement to be added are checked first.

Exploring the neighborhood. With the aim of testing the flexibility of our approach, we have tried two different strategies for the exploration of the neighborhood. This can be done just by providing two different implementations for the (required) choose_best_move() method.

(a) In the *direct exploration of the neighborhood*, the set of items to be dropped from the knapsack is selected first, and then, the space left by the dropped items is used to add new items. The selection of the items to be dropped is described in Algorithm 2 (see Appendix A.5). To check the capacity constraints, the algorithm uses a vector $b^{r'}$ (which initially is a copy of b^r) that contains the variations in the capacity due to the items that are added and dropped. At every iteration, the algorithm determines in which dimension the knapsack is saturated the most (i.e., the argument i^* minimizing the remaining capacity $b^{r'}[i]$.) Then, in this dimension, the item with the worst relation between its benefit and its cost will be candidate to be dropped, provided it is not tabu. In case all the items in the knapsack are tabu, an item is selected according to the aspiration criteria. However, if the knapsack becomes empty, the search is interrupted and the number of items is set to one; thus, at least one item will always be removed from the knapsack.

The selection of the items to be added is done iteratively according to Algorithm 3 (see Appendix A.5). The candidate items to be considered cannot be already included in the current configuration of the knapsack and cannot be tabu. From those candidates the algorithm will choose, at each iteration, the item providing the higher benefit that does not violate the capacity constraints. When an item l^* is chosen to be added to the movement, the vector $b^{r'}$ is updated by decreasing the free capacity at each dimension according to the cost of the item at that dimension. In case the element l^* is tabu, the aspiration criteria decides whether the item is added or not.

(b) The *combinatorial exploration of the neighborhood* allows to explore the neighborhood in order and more exhaustively. It differs from the direct exploration in two main points: first, the movements only imply a change in the state of one or two items of the knapsack, and second, the best movement is chosen among all those movements that can be generated combinatorially from two solution components, i.e., from two items, and that would lead to feasible solutions in the neighborhood of the current solution. A movement is here composed either by two items (k, l), one already in the knapsack and the other one not, which will exchange their state, or just by one item (k) not in the knapsack. From all the movements leading to feasible solutions, we select the one providing the greatest improvement in the fitness, which does not violate any of the constraints and is not tabu (or has aspiration, in case all of them are tabu).

This strategy is more expensive as compared to the direct exploration (quadratic time versus linear time of the direct exploration) due to the exhaustive generation of all the combinations of pairs of items.

Running the instantiation. Once the implementation of all the required classes is completed in order to instantiate the 0–1MKNP problem, we can run the skeleton in different settings, namely, sequential or parallel with the IR, IRAS, MS or MSNP models. To this aim, the user only needs to declare an object of the appropriate sub-class of the `Solver` class (i.e., `Solver_Seq`, `Solver_IR`, `Solver_IRAS`, `Solver_MS` or `Solver_MSNP`), and call the `run` method in it (see an example in Fig. 11.)

```
#include "TabuSearch-MKNP.hh"                // header file of the instantiation

int Main () {
    using skeleton TabuSearch;
    Problem pbl;  cin >> pbl;                // read the problem instance
    Setup params; cin >> params;             // read the setup parameters
    Solver_IR solver(pbl,params);            // select the IR solver
    solver.run();                            // run the solver
    cout << solver.best_solution() << endl;  // report the best solution found
    cout << solver.best_cost() << endl;      // report its cost
}
```

Figure 11: Example to run the instantiation for the 0–1MKNP problem in parallel using the IR model.

6.3 Experimental evaluation

Measuring the performance of a meta-heuristic implementations requires testing on a large set of real size instances. Moreover, finding appropriate values for the search parameters of the meta-heuristic is almost indispensable.

We take profit of the advantages that parallelism offers us in terms of reduction of the computation time, and use the IRAS model introduced in Sec. 5 for the fine tuning of parameters. Furthermore, for our purpose of measuring the performance of the implementation for the 0–1MKNP problem, we also use the other parallel models introduced in Sec. 5, as well as the sequential implementation, for experimenting.

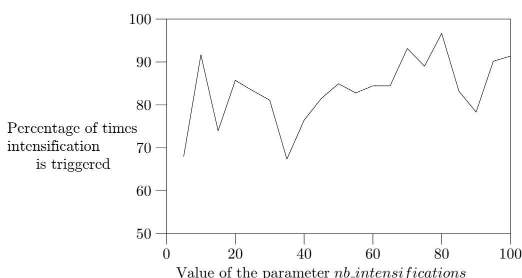
6.3.1 Parameter tuning

Many mutually-dependent parameters are involved in the instantiation of the TS skeleton for the 0–1MKNP problem. Some of these parameters are related to the instantiation proposed while some others are proper to the TS meta-heuristic. As for the parameters of the TS, the basic parameters are those controlling the stopping conditions (`max_execution_time` and `independent_runs`) and the influence of the historical search memory (`tabu_list_size`). Other important parameters are used to control the search process, specially the neighborhood exploration (`max_neighbors`), the intensification (`max_repetitions`, `nb_best_sols` and `nb_intensification`) and the diversification (`history_rep` and `nb_diversifications`). In the following, we report the best values obtained in the tuning for each of these parameters (see Table 1 for a summary).

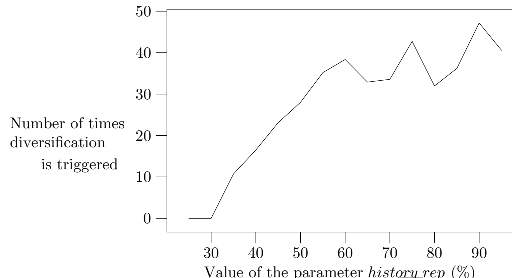
`max_execution_time` and `independent_runs`: We found that 2 independent runs of the program already provided relatively good solutions (see, e.g., the average results in Tables 3 and 4 for 2 processors), although it depends strongly on the number of iterations of each run. The number of iterations required depends on the size of the problem, specially on n. A value of 900 seconds for this parameter already provides good solutions, even for big size instances.

Table 1: Description of the parameters.

Parameter	Description
`max_execution_time`	Maximum execution time of each independent run of the algorithm.
`independent_runs`	Number of independent runs.
`tabu_size`	Size of the tabu list (for the short term memory of the TS method).
`max_neighbors`	Maximum number of neighbors to explore.
`max_repetitions`	Number of non-improvements before the intensification is triggered.
`nb_intensifications`	Number of iterations that the intensification lasts.
`nb_diversifications`	Number of iterations that the soft diversification lasts.
`nb_best_sols`	Number of best solutions maintained as history for the intensification.
`history_rep`	How often items can belong to a solution before the diversification.



(a) Influence of `nb_intensifications` on the intensification.



(b) Influence of `history_rep` on the diversification.

Figure 12: Influence of parameters on intensification and diversification.

`tabu_size`**:** We have found that a tabu list size of length in $[3, 15]$ performs good. The final value is chosen randomly in this interval.

`max_neighbors`**:** No specific value could be decided for this parameter from our experiments. Therefore, we decided to explore the whole neighborhood instead of only a portion of it.

`max_repetitions`**:** The value of this parameter is in charge of starting the intensification process. A too low value can make the intensification to start early while a too high value delays the intensification. Since it is very strict and unlikely that exactly the same solution is obtained in consecutive iterations, we have relaxed the definition of difference between solutions by considering the Hamming distance between them. Then, two solutions are considered equal if they have at most $\alpha n / \ln n$ $(0 < \alpha \leq 1)$ differences. Experimentally, good results are obtained for $\alpha = 0.1$.

`nb_best_sols`**:** The bigger the values for this parameter, the more historical information available for a good intensification. However, this also makes the intensification processes slower. The experiments suggest that keeping between 10 and 15 solutions is a good equilibrium between the benefit of the intensification and its computational cost.

`nb_intensifications`**:** We have observed in the tuning process that even small values of this parameter are useful, thus justifying the need for intensification. Again we had to find a good trade-off between the computational cost and the contribution of the intensification to the quality of solutions. According to the experiments (see Fig. 12(a)), a high percentage of success for intensification can be obtained for the low value of parameter close to 10.

`history_rep` is used to decide whether an item has been incorporated many times to the solution during the search (when so, the soft diversification is triggered). In order to assure a good long-term influence of this parameter and to overcome the drawbacks found when using the constant value in [30], we internally use an another parameter based on this one. We define `rep` $=$ (`history_rep`$/100) \cdot \max\{$`history`$[j], j = 1, \ldots, n\}$, which gets a proportional value (w.r.t. the elements of the array `history`) at each iteration of the diversification. The better

results are obtained for high values of this parameter (between 80% and 95%, see Fig. 12(b)).

`nb_diversifications`: The experiments performed could not clarify which should be a good value for this parameter. However, we could observe that worse quality solutions were obtained when no diversification is performed, i.e., when the parameter is 0. From all the values we tried out, fixing it to 10 showed to provide reasonable good solutions.

6.3.2 Computational results

After tuning the parameters involved in TS for the 0–1MKNP, we have tested more in detail the instantiation proposed in Sec. 6.2 for the problem. We report results concerning the sequential as well as the parallel implementations provided by the TS skeleton. In order to obtain some statistical significance about the robustness of the implementation, the same experiment was run several times and thus the results refer to the average results obtained. In our experiments, each instance is run 20 times (i.e., the `independent_runs` parameter is set to 20), and each of the 20 runs is fixed to last 900 seconds (i.e., the `max_execution_time` parameter is set to 900).[3] The setting for the remaining parameters is the same for each experiment, as obtained in the tuning (see Sec. 6.3.1). All our experiments are run in a cluster of nine AMD K6-11 computers with 450 MHz processors and 256Mb of memory. The implementations are done in C++ using LEDA [28], MPI and are compiled with GCC 2.95.2.

We tested small, medium and big size instances (in terms of the number of variables) taken from the literature. Small instances ($n \leq 50$) are taken from [18, 10], middle size instances ($50 < n \leq 100$) are taken from [40, 33], and big size instances ($100 < n \leq 500$) are taken from the OR-library [3]. We report here results concerning some of their big size instances. More results concerning smaller and very big-size instances are included as appendix.

Sequential executions. Table 2 summarizes some of the results obtained from the sequential implementation. We report results for the direct and the combinatorial neighborhood exploration strategies described in Sec. 6.2.

We notice that the average deviation of best solutions w.r.t. the optimum is very small. On the one hand this shows that our instantiation of the problem is appropriate and, on the other, the small values for the deviation also indicates the robustness of our approach in the sense that the values we found in the tuning process perform very well for a large set of different instances. Observe, however, that the combinatorial exploration provides better results. Therefore, we choose this latter strategy for the experiments concerning parallel executions.

Parallel executions. Table 3 summarizes the results obtained for the IR and IRAS parallel implementations. We can observe that, when fixing the same computation time, the IR model performs better than the IRAS model. This is due to the fact that, in the same amount of time, the IRAS model cannot do as many iterations of the algorithm as the IR model can do. The IRAS model needs also more extra time for generating the strategies and for communication. Moreover, some of those generated strategies may not be appropriate, although the IRAS model invests time on exploring with them. However, we have observed in additional experiments (not reported here) that the effect of this drawback is reduced when the execution time is increased. The nature of the IR and IRAS models reinforces the general observation that the more processors are used for computation, the better results we obtain, since a broader area of the search space is explored.

Table 4 summarizes the results obtained for the execution of the MS and MSNP parallel implementations. We can observe that, for a small number of processors, the MS model achieves better results than the MSNP model. Again, a reason for this effect might be the fact that, in the fixed execution time the MS model does usually more iterations than the MSNP does. However, due to communication overhead, this effect fades away when more processors are used because, in proportion, more time is used in communication. Concerning the MS model, it is worth mentioning

[3] In the literature (see [8]), the execution times vary roughly from 700 seconds (for quite small instances) to 2500 seconds.

that the results obtained with eight processors are not better than the ones obtained with four processors. More processors in the MS model represent more exploration of the neighborhood, but also more time and more re-exploration of already explored areas; in a fixed time, this translates into less iterations of the algorithm and therefore worse results. This effect does not appear in the MSNP model, since there the usage of more processors implies a thorougher exploration of each neighborhood. Therefore, the more processors participating in the MSNP model, the better results are obtained. Moreover, the more processors, the less portion of the neighborhood that each of them will have to explore, and thus, the faster a neighborhood will be explored. This is the reason why, when using eight processors and a fixed amount of time, the number of iterations performed in both models becomes similar.

7 Conclusions

In this work we have presented an approach to obtain a generic implementation of the Tabu Search meta-heuristic by using algorithmic skeletons. Algorithmic skeletons constitute a way to reduce the effort to develop sequential and parallel applications. Indeed, the skeleton separates the implementation of the generic algorithm from the specific knowledge of the concrete problem being solved. The implementation of the generic algorithm is provided in the skeleton and the user is just required to describe the elements defining his problem of interest. Moreover, due to this separation of concerns, we are able to encapsulate in the skeleton not only the generic algorithm but also different parallel implementations of it. There is no need for the user to know parallel programming: an important advantage of our approach is the use of sequential constructs in the interface of the skeleton and its instantiation. The result is a program able to run in both sequential and parallel settings.

We have instantiated our skeleton for several well known combinatorial optimization problems for which TS has already been implemented. In this process, we have observed that our approach shows interesting properties regarding developing time, flexibility and easiness of use, quality of solutions and computation efficiency.

In this paper, we have chosen to exemplify in detail our approach to instantiate the 0–1 Multidimensional Knapsack problem. From this instantiations, we have reported extensive experimental results from standard benchmarks for this problem. The parallel program has been executed on a cluster of commodity machines and uses MPI as a communication library. In spite of the genericity of our implementation, our results show that the resulting program provides high quality solutions very close to the optimal ones. Moreover, we have shown the flexibility of our approach by providing two different implementations for the problem by just changing the neighborhood implementation.

Acknowledgments

The authors wish to thank the members of the MALLBA Project for their support and useful discussions while conducting this research. We are also grateful to Christian Blum for providing us with the sources of the Fig. 3 from his article [7].

Table 2: Results obtained in the **sequential execution of big-size instances** of the OR-library when using two different implementations of the neighborhood exploration for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in 900s.

				Direct exploration of the neighborhood				Combinatorial exploration of the neighborhood			
Instance	n	m	Best known cost	Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
OR5x250-00	250	5	59312	55963	55450.8	0.0565	94054.8	**58900**	58469.8	0.0069	1970.8
OR5x250-29	250	5	154662	153312	153257.2	0.0087	115364.0	**154309**	153998.8	0.0023	1402.6
OR10x250-00	250	10	59187	56213	55945.6	0.0502	55132.9	**58076**	57680.8	0.0188	1659.6
OR10x250-29	250	10	149704	148342	148121.9	0.0091	68679.3	**148868**	148292.6	0.0056	901.4
OR30x250-00	250	30	56693	54711	54534.6	0.0350	15215.1	**55946**	55490.2	0.0132	1672.2
OR30x250-29	250	30	149572	148588	148349.6	0.0066	22437.6	**148761**	148579.0	0.0054	768.6

Table 3: Results obtained in the **IR and IRAS parallel models' execution of big-size instances** of the OR-library (using a combinatorial exploration of the neighborhood) for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in 900s.

					IR model				IRAS model			
Instance	n	m	Best known cost		Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
				2 procs.	58610	58492.0	0.0118	2163.0	57645	57223.8	0.0281	1238.4
OR5x250-00	250	5	59312	4 procs.	**58908**	58636.8	0.0068	6784.8	58417	57698.8	0.0151	3192.0
				8 procs.	58833	58739.2	0.0081	15079.0	58106	57713.2	0.0203	8922.8
				2 procs.	154437	154154.6	0.0015	1452.8	153871	152940.8	0.0051	782.8
OR5x250-29	250	5	154662	4 procs.	**154446**	154132.3	0.0014	3362.8	153666	153297.0	0.0064	1360.6
				8 procs.	154437	154339.0	0.0015	8173.6	154188	153972.0	0.0031	4407.8
				2 procs.	58353	58005.2	0.0141	1837.0	56513	55685.0	0.0452	956.4
OR10x250-00	250	10	59187	4 procs.	58090	58015.6	0.0185	5204.4	56376	55757.0	0.0475	1566.4
				8 procs.	**58481**	58301.6	0.0119	13605.0	56496	56198.8	0.0455	4435.4
				2 procs.	148499	148056.2	0.0080	822.0	147781	147739.4	0.0128	416.6
OR10x250-29	250	10	149704	4 procs.	149009	148767.0	0.0046	2782.6	148077	147831.6	0.0109	1152.8
				8 procs.	**149058**	148889.0	0.0043	5717.2	149030	148840.2	0.0045	3762.8
				2 procs.	55567	55501.8	0.0199	2016.8	55346	54872.2	0.0238	674.2
OR30x250-00	250	30	56693	4 procs.	55909	55648.2	0.0138	4404.8	55412	55055.2	0.0226	1575.6
				8 procs.	**55959**	55811.0	0.0129	10253.0	55725	55494.6	0.0170	4764.8
				2 procs.	148710	148489.0	0.0058	555.2	148147	147761.8	0.0095	223.2
OR30x250-29	250	30	149572	4 procs.	148901	148621.0	0.0045	1326.8	148010	147720.4	0.0104	398.0
				8 procs.	149024	148886.6	0.0037	4785.0	**149034**	148969.6	0.0036	2231.4

Table 4: Results obtained in the **MS and MSNP parallel models' execution of big-size instances** of the OR-library (using a combinatorial exploration of the neighborhood) for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in 900s.

					MS model				MSNP model			
Instance	n	m	Best known cost		Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
OR5x250-00	250	5	59312	2 procs.	58389	58282.2	0.0156	1711.8	56258	55362.2	0.0515	271.0
				4 procs.	**58736**	58318.6	0.0097	1389.4	57838	57059.6	0.0249	747.2
				8 procs.	58660	58274.2	0.0110	771.0	58085	57923.0	0.0207	1223.8
OR5x250-29	250	5	154662	2 procs.	**154398**	154199.6	0.0017	1101.4	152793	152473.6	0.0121	252.2
				4 procs.	154315	153862.0	0.0022	838.6	153518	153079.8	0.0074	484.4
				8 procs.	154105	153643.0	0.0036	554.8	153636	153283.2	0.0066	746.4
OR10x250-00	250	10	59187	2 procs.	**58176**	58097.6	0.0171	1431.0	54510	54135.6	0.0790	114.0
				4 procs.	57847	57606.6	0.0226	1141.6	55352	54912.8	0.0648	278.4
				8 procs.	57674	57227.4	0.0256	492.4	55374	54941.8	0.0644	505.4
OR10x250-29	250	10	149704	2 procs.	**148532**	148140.0	0.0078	674.8	147777	147590.0	0.0129	209.4
				4 procs.	148426	148237.2	0.0085	625.6	147865	147771.0	0.0123	363.6
				8 procs.	148151	147805.2	0.0104	368.2	148220	147954.2	0.0099	669.2
OR30x250-00	250	30	56693	2 procs.	55642	55285.8	0.0185	797.0	54317	53820.0	0.0419	95.0
				4 procs.	**56037**	55174.6	0.0116	501.2	55082	54112.4	0.0284	142.8
				8 procs.	55323	54937.8	0.0242	234.5	54951	54386.4	0.0307	269.0
OR30x250-29	250	30	149572	2 procs.	**148934**	148416.2	0.0043	424.6	147155	147111.6	0.0162	57.0
				4 procs.	148862	148598.8	0.0047	334.2	148142	148019.2	0.0096	129.6
				8 procs.	148582	148259.4	0.0066	199.0	148424	148099.2	0.0077	221.4

References

[1] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A library of skeletons for combinatorial optimisation. volume 2400 of *LNCS*, pages 927–932. Springer, 2002.

[2] E. Alba and J. F. Chicano. Solving the error correcting code problem with parallel hybrid heuristics. In *Proceedings of ACM SAC'04*, volume 2, pages 985–989, 2004.

[3] J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. Publicaly available at http://people.brunel.ac.uk/∼mastjjb/jeb/info.html.

[4] M. Blesa, L. Hernàndez, and F. Xhafa. Parallel skeletons for Tabu Search method. In *8th International Conference on Parallel and Distributed Systems*, pages 23–28. IEEE Computer Society Press, 2001.

[5] M. Blesa, L. Hernàndez, and F. Xhafa. Parallel skeletons for Tabu Search method based on search strategies and neighborhood partition. volume 2328 of *LNCS*, pages 185–193. Springer, 2002.

[6] C. Blum and M. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers & Operations Research*, 2004. In press.

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

[8] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.

[9] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, MA, 1989.

[10] C. Cotta and J. Troya. A hybrid genetic algorithm for the 0-1 multiple knapsack problem. In *3rd International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 251–255. Springer, 1998.

[11] T. Crainic and M. Toulouse. *Parallel Metaheuristics*. Kluwer Academic Publishers, 2004.

[12] T. Crainic, M. Toulouse, and M. Gendreau. Towards a taxonomy of parallel Tabu Search heuristics. *INFORMS Journal of Computing*, 9(1):61–72, 1997.

[13] J. Crotinger. Generic programming in the POOMA framework. volume 1766 of *LNCS*. Springer, 2000.

[14] M. Dell'Amico and M. Trubian. Applying Tabu Search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1986.

[15] M. Esö, L. Ladányi, T. Ralphs, and L. Trotter Jr. Fully parallel generic branch-and-cut framework. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[16] C. Fiechter. A parallel Tabu Search algorithm for large travelling salesman problem. *Discrete Applied Mathematics*, 51:243–267, 1994.

[17] R. Francis and J. White. *Facility Layout and Location*. Prentice-Hall, 1974.

[18] A. Freville and G. Plateau. Hard 0-1 multiknapsack test problems for size reduction methods. *Investigation Operativa*, 1:251–270, 1990.

[19] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.

[20] J. Gerlach and M. Sato. Generic programming for parallel mesh problems. volume 1732 of *LNCS*, pages 108–119. Springer, 1999.

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

[22] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[23] C. Hughes and T. Hughes. *Parallel and Distributed Programming Using C++*. Addison-Wesley, first edition, 2003.

[24] K. Jörnsten and A. Løkketangen. Tabu Search for weighted k-cardinality trees. *Asia-Pacific Journal of Operations Research*, 14(2):9–26, 1997.

[25] J. Krarup and P. Pruzan. Computer-aided layout design. *Mathematical Programming Study*, 9:75–94, 1978.

[26] M. Laguna, J. Barnes, and F. Glover. Tabu Search methodology for a single machine scheduling problem. *Journal of International Manufacturing*, 2:63–74, 1991.

[27] MALLBA Project. http://www.lsi.upc.es/∼mallba.

[28] K. Mehlhorn and S. Näher. *LEDA. A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. www.algorithmic-solutions.com/enleda.htm.

[29] Z. Michalewicz and D. Fogel. *How to solve it: modern heuristics*. Springer, 2000.

[30] S. Niar and A. Freville. A parallel Tabu Search algorithm for the 0-1 Multidimensional Knapsack problem. In *11th International Parallel Processing Symposium*, pages 512–516, 1997.

[31] C. Porto and C. Ribeiro. Parallel Tabu Search message-passing synchronous strategies for task scheduling under procedence constraints. *Journal of Heuristics*, 1(2):207–223, 1996.

[32] S. Porto and C. Ribeiro. A Tabu Search approach to task scheduling on heterogeneous processor under precedence constraints. *International Journal of High-Speed Computation*, 7(2):45–71, 1995.

[33] S. Senyu and Y. Toyoda. An approach to linear programming with 0-1 variables. *Management Science*, 15:196–207, 1968.

[34] J. Siek, A. Lumsdaine, and L.-Q. Lee. Generic programming for high performance numerical linear algebra. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM Proceedings in Applied Mathematics, SIAM Press, 1998.

[35] J. Skorin-Kapov. Tabu Search applied to the Quadratic Assignment Problem. *ORSA Journal on Computing*, 2(1):33–45, 1990.

[36] L. Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.

[37] E. Taillard. Robust Tabu Search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[38] E. Taillard. Parallel iterative search methods for vehicle routing problem. *Networks*, 23:661–673, 1993.

[39] L. Trotter. Generic parallel implementation for integer programming. In *DONET Spring School on Computational Combinatorial Optimization*, 2000.

[40] H. Weingartner and D. Ness. Methods for the solution of the multi-dimensional 0/1 knapsack problem. *Operations Research*, 15:83–103, 1967.

[41] M. Widmer. The job-shop scheduling with tooling constraints: A Tabu Search approach. *Journal of the Operational Research Society*, 42:75–82, 1991.

A Details on the instantiation of the 0-1MKNP problem

A.1 The Problem class

Header file

```
requires class Problem {
public: ...
private:
    int nb_items;
    int nb_constraints;
    array2<integer> constraints;
    array<integer>  benefits,capacities;
}
```

Implementation file

```
Problem::Problem ()
: nb_items(0),  nb_constraints(0),
  constraints(nb_constraints,nb_items),
  benefits(nb_items), capacities(nb_constraints) {}

Direction Problem::direction () const {
    return maximization;
}
```

A.2 The Movement class

Header file

```
enum Action {add=1, drop=0};

requires class Movement {
public: ...
private:
    array<int> items;
    int nb_drop,nb_add,tabulife;
}
```

Implementation file

```
Movement::Movement (const Problem& pbm) {
    items=array<int>(pbm.nb_items);
    nb_drop=nb_add=tabulife=0;
}

void Movement::can_apply (int item, const Action A) {
    items[nb_drop+nb_add]=item;
    if (A==drop) nb_drop++; else nb_add++;
}

bool Movement::belong (int item) {
    int  idx=0;
    while (idx<(nb_drop+nb_add) and items[idx++]!=item) {}
    return idx>0 and items[--idx]==item;
}
```

A.3 The TabuStorage class

Header file

```
requires class TabuStorage {
public: ...
private:
    array<int>      tl,history;
    array<Solution> best_solutions;
    array<double>   item_frequencies;
    int             history_rep;
    int             max_best_solutions;
    int             nb_best_solutions;
}
```

Implementation file

```
TabuStorage::TabuStorage (const SetUpParams& setup,
                          const Problem& pbm)
: config(setup), problem(pbm),
  tl(problem.nb_items), history (problem.nb_items),
  history_rep (setup.history_rep),
  best_solutions(setup.nb_best_sols),
  max_best_solutions(setup.nb_best_sols),
  nb_best_solutions(0),
  item_frequencies(problem.nb_items) {
    tl.init(MINUS_INFINITY);
    history.init(0);
    item_frequencies.init(0);
    best_solutions.init(Solution(pbm));
    start_random();
}

void TabuStorage::make_tabu (Movement& mov,
           const Solution& sol, const State& state) {
    for (int i=mov.nb_drop+mov.nb_add-1; i>=0; i--) {
        tl[mov.items[i]]=state.current_iteration;
        if (i>=mov.nb_drop) history[mov.items[i]]++;
    }
    update_best_sols(sol);
}

bool TabuStorage::is_tabu (const Movement& mov,
                  const Solution& sol, const State& state) const {
    int iter=state.current_iteration;
    for (int i=mov.nb_add+mov.nb_drop-1; i>=0; ) {
        if (tl[mov.items[i--]]+config.tabulife>=iter) return true;
    }
    return false;
}
```

A.4 The `Solution` class

Header file

```
const double PENALTY = 0.5;
const double PREMIUM = 2;
const double MINFREQ_REWARD = 0.9;
const double MAXFREQ_PENALIZE = 0.1;

typedef int SolutionComponent;

requires class Solution {
public: ...
private:
    array<int>   contents;
    Problem&     problem;
    TabuStorage* user_data;
}
```

Implementation file

```
void Solution::initial_solution() {
    array<int>    emptyknapsack(problem.nb_items);
    array<double> sorted_benefits;
    array<int>    sorted_indices(problem.nb_items);
    sorted_benefits = problem.benefits;
    contents        = emptyknapsack;

    for (int j=0; j<problem.nb_items; j++)
        sorted_indices[j]=j;

    sort_benefits(sorted_benefits,sorted_indices,0,
                  problem.nb_items-1);

    for (int j=problem.nb_items-1; j>=0; j--) {
        int idx=sorted_indices[j];
        if (!check_capacities(idx)) {
            contents[idx]=0;
        } else {
              contents[idx]=1;
              update_capacities(idx,add);
}   }   }

SolutionComponent& Solution::operator[] (int i) {
    return contents[i];
}

double Solution::cost () const {
    double cost=0;
    for (int item=0;item<problem.nb_items;item++)
        cost+=problem.benefits[item]*contents[item];
    return cost;
}

double Solution::delta (const Movement& mov) const {
    double differential=0;
    int idx_drop=0, idx_add=0, item;
    while(idx_drop<mov.nb_drop) {
        item=mov.movement[idx_drop++];
        differential-=problem.benefits[item];
    }
    while(idx_add<mov.nb_add) {
        item=mov.movement[idx_drop+idx_add++];
        differential+=problem.benefits[item];
    }
    return differential;
}

void Solution::apply (const Movement& mov) {
    int idx_drop=0, idx_add=0, item;
    while(idx_drop<mov.nb_drop) {
        item=mov.movement[idx_drop++];
        contents[item]=0;
    }
    while(idx_add<mov.nb_add) {
        item=mov.movement[idx_drop+idx_add++];
        contents[item]=1;
}   }

int size () const {
    return contents.size();
}

void Solution::reward() {
    user_data->build_frequencies();
    for (int item=0; item<problem.nb_items; item++) {
        double freq = user_data->item_frequencies[item];
        if(freq>=MIN_FREQ_REWARDED) {
            problem.benefits[item]*=PREMIUM;
            rewarded.insert(item);
        } else if (freq<=MAX_FREQ_PENALIZED) {
            problem.benefits[item]*=PENALTY;
            penalized.insert(item);
}   }   }

void Solution::escape() {
    solution.init(0);
    int item=problem.nb_items-1;
    while(item>=0) {
        int randnum = get_random(0,1);
        contents[item--] = randnum==1 and check_capacities(item));
}   }

void Solution::penalize() {
    set<int>  diverItems;
    Movement  mov(problem);
    int       tadded=0;
    bool      itemFits=false;
    double    hist;

    for(int item=problem.nb_items-1; item>=0; item--) {
        if(user_data->history[item]>tadded) {
            tadded=user_data->history[item];
    }   }
    int rep=user_data->history_rep/100*tadded;
    for(int item=problem.nb_is-1; item>=0; item--) {
        hist = user_data->history[item];
        if (contents[item]==1 and hist>rep) {
            mov.can_apply(item,drop);
            diveris.insert(item);
    }   }

    for (int item=problem.nb_items-1; item>=0; item--)  {
        hist = user_data->history[item];
        if (contents[item]==0 and hist<=rep)  {
            apply(mov);
            itemFits=check_capacities(item);
            unapply(mov);
            if (itemFits) {
                mov.can_apply(item,add);
                diverItems.insert(item);
    }   }   }
    apply(mov);
    user_data->diver_make_tabu(diverItems);
}
```

A.5 Algorithms for the direct exploration of the neighborhood

For the sake of simplicity and space-saving, we just describe in pseudo-code how the selection of the items to be dropped and added is is implemented when performing a direct exploration of the neighborhood.

Algorithm 2 : Selection of the items to be dropped in the direct exploration of the neighborhood

Choose uniformly at random nb_drop in $\{1 \ldots 5\}$
$b^{r'} \leftarrow b^r$, $l \leftarrow 0$, continue $\leftarrow$ TRUE
while (l <nb_drop) **and** continue **do**
  Find the most saturated constraint $i^* \leftarrow \text{argmin}\{b^{r'}[i], i = 1 \ldots m\}$
  Find the 'worse' item j^* in the dimension i^* which is not tabu,
    $j^* \leftarrow \text{argmax}\{a[i^*, j]/c[j],\ j = 1 \ldots m \mid x[j] = 1, j \notin \text{tl}\ \}$
  if j^* exists **then**
    Add j^* to the movement as 'to be dropped'
    Update $b^{r'}$
    $l \leftarrow l + 1$
  else
    continue $\leftarrow$ FALSE
  end if
end while
nb_drop $\leftarrow \min\{\text{nb_drop}, 1\}$

Algorithm 3 : Selection of the items to be added in the direct exploration of the neighborhood

nb_add $\leftarrow$ 0, continue $\leftarrow$ TRUE
while continue **do**
  Find the best non-tabu candidate l^*, i.e.,
    $l^* \leftarrow \text{argmax}\{c[l], l = 1 \ldots n \mid x[l] = 0, l \notin \text{tl},\ a[i, l] \leq b^{r'}[l],\ \forall i = 1 \ldots m\}$
  if l^* exists **then**
    Add l^* to the movement as 'to be added'
    $b^{r'}[i] \leftarrow b^{r'}[i] - a[i, l^*], \forall i = 1 \ldots m$
    nb_add $\leftarrow$ nb_add+1
  else
    continue $\leftarrow$ FALSE
  end if
end while

B Additional experimental results

Table 5 reports some of the results obtained in the sequential execution of different size instances when using two different implementations of the neighborhood exploration for the 0–1MKNP problem. One can observe that, clearly, a combinatorial exploration obtains better results in most of the cases than a direct exploration of the neighborhood.

Table 6 reports additional results on the IR and MS models for some (not so) small instances of the OR-library. One can observe that, in general, is better to use a parallel model based on independent runs than a master-slave model. Observe also that, although being instances of small size, often more than two processors are needed in order to obtain better results.

Table 7 reports additional results on the IR and MS models for some very big instances of the OR-library. One can observe that, for very big instances it is more beneficial to use more processors in parallel and that, in general, is better to use a parallel model based on independent runs than a master-slave model. The reason for that is the big amount of communication time (and thus not computation time) that the latter consumes.

Table 5: Results obtained in the **sequential execution of different size instances** of the OR-library when using two different implementations of the neighborhood exploration for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in the established execution time. The established time was 600 seconds for those instances with $n = 100$, 900 seconds for those instances with $n = 250$, and 1200 seconds for those ones with $n = 500$. For the sake of comparability, the results in Table 2 are also reported here (middle part).

				Direct exploration of the neighborhood				Combinatorial exploration of the neighborhood			
Instance	n	m	Best known cost	Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
OR5x100-00	100	5	24381	22781	22615.1	0.0656	158620.2	**24381**	24238.8	0	10441.6
OR5x100-29	100	5	59965	59639	59496.7	0.0054	179516.0	**59945**	59855.2	0.0003	6902.4
OR10x100-00	100	10	23064	22478	22360.4	0.0254	85629.8	**22944**	22822.0	0.0052	10340.4
OR10x100-29	100	10	60633	60629	60518.9	0.0001	68679.3	**60633**	60592.6	0	9335.6
OR30x100-00	100	30	21946	21614	21520.7	0.0151	31615.7	**21719**	21656.8	0.0103	5362.0
OR30x100-29	100	30	60603	60432	60292.8	0.0028	37874.5	**60603**	60349.6	0	4338.8
OR5x250-00	250	5	59312	55963	55450.8	0.0565	94054.8	**58900**	58469.8	0.0069	1970.8
OR5x250-29	250	5	154662	153312	153257.2	0.0087	115364.0	**154309**	153998.8	0.0023	1402.6
OR10x250-00	250	10	59187	56213	55945.6	0.0502	55132.9	**58076**	57680.8	0.0188	1659.6
OR10x250-29	250	10	149704	148342	148121.9	0.0091	68679.3	**148868**	148292.6	0.0056	901.4
OR30x250-00	250	30	56693	54711	54534.6	0.0350	15215.1	**55946**	55490.2	0.0132	1672.2
OR30x250-29	250	30	149572	148588	148349.6	0.0066	22437.6	**148761**	148579.0	0.0054	768.6
OR5x500-00	500	5	120130	112991	112575.3	0.0594	68692.6	**116969**	115622.4	0.0263	426.2
OR5x500-29	500	5	299904	296939	296079.3	0.0099	81719.4	**297013**	295104.6	0.0096	295.4
OR10x500-00	500	10	117726	111773	111486.7	0.0506	35487.5	**115919**	114188.2	0.0153	592.2
OR10x500-29	500	10	307014	**303943**	303642.9	0.0100	44267.8	303417	302680.8	0.0117	205.0
OR30x500-00	500	30	115868	111272	110942.4	0.0397	10459.3	**112898**	112028.0	0.0256	759.6
OR30x500-29	500	30	300460	**298719**	298533.4	0.0058	12410.6	298634	297754.4	0.0061	170.0

Table 6: Results obtained in the **IR and MS parallel models' execution of some (not so) small instances** of the OR-library (using a combinatorial exploration of the neighborhood) for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in 600s.

					IR model				MS model			
Instance	n	m	Best known cost		Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
				2 procs.	**24381**	24075.2	0	9112.0	24220	24109.2	0.0066	5459.6
OR5x100-00	100	5	24381	4 procs.	24329	24244.6	0.0021	25710.0	24329	24232.4	0.0021	4169.8
				8 procs.	24329	24252.2	0.0021	58314.8	24282	24052.2	0.0041	1838.0
				2 procs.	59931	59728.0	0.0006	5562.6	59955	59879.2	0.0002	5691.8
OR5x100-29	100	5	59965	4 procs.	59955	59919.4	0.0002	24355.2	59846	59738.6	0.0020	3294.8
				8 procs.	**59965**	59939.4	0	54557.8	59896	59792.6	0.0012	1723.0
				2 procs.	22978	22851.6	0.0037	10430.6	**23055**	22931.4	0.0004	4926.0
OR10x100-00	100	10	23064	4 procs.	23011	22931.0	0.0023	23355.8	22966	22833.4	0.0042	3177.4
				8 procs.	**23055**	23022.4	0.0004	61070.0	22717	22670.0	0.0150	1079.8
				2 procs.	**60633**	60590.0	0	6994.2	60629	60579.8	0.0001	3970.6
OR10x100-29	100	10	60633	4 procs.	**60633**	60553.6	0	18109.0	**60633**	60590.2	0	2770.8
				8 procs.	**60633**	60597.8	0	59533.2	60629	60507.4	0.0001	1040.4
				2 procs.	21829	21664.0	0.0053	6749.4	21707	21629.4	0.0109	2240.0
OR30x100-00	100	30	21946	4 procs.	**21946**	21780.2	0	17297.4	21662	21620.8	0.0129	1337.8
				8 procs.	**21946**	21827.4	0	38477.4	21616	21493.8	0.0150	430.4
				2 procs.	60472	60381.6	0.0022	4580.0	60327	60158.0	0.0046	1700.2
OR30x100-29	100	30	60603	4 procs.	**60603**	60410.8	0	11961.6	60351	60160.4	0.0042	1104.4
				8 procs.	60554	60493.2	0.0008	31864.0	60123	60072.2	0.0079	387.2

Table 7: Results obtained in the **IR and MS parallel models' execution of very big-size instances** of the OR-library (using a combinatorial exploration of the neighborhood) for the 0–1MKNP problem. Each instance is run 20 times. Instance information is found in the first four columns; the 5th and 9th columns report the best cost obtained; the 6th and 10th report the average cost obtained and the 7th and 11th the deviation of the best obtained costs w.r.t. the Best known cost. The 8th and 12th columns indicate the number of iterations that the algorithm performed in 1200s.

					IR model				MS model			
Instance	n	m	Best known cost		Best obtained cost	Average cost	dev. from opt.	Iterations	Best cost	Average cost	dev. from opt.	Iterations
OR5x500-00	500	5	120130	2 procs.	116958	114851.2	0.0264	353.6	116025	115367.4	0.0342	373.2
				4 procs.	116561	113983.6	0.0297	1219.4	116550	114838.4	0.0298	378.4
				8 procs.	**117257**	114962.2	0.0239	2669.8	115552	112096.8	0.0381	281.4
OR5x500-29	500	5	299904	2 procs.	295678	294292.4	0.0141	246.6	295162	294688.0	0.0158	233.2
				4 procs.	**296956**	295761.0	0.0098	813.0	296891	293510.8	0.0100	234.6
				8 procs.	296631	295473.4	0.0110	1751.8	295432	293429.4	0.0149	191.0
OR10x500-00	500	10	117726	2 procs.	115191	112998.6	0.0215	609.6	115087	113584.4	0.0224	592.2
				4 procs.	115241	114704.0	0.0211	1740.0	114635	113513.6	0.0263	436.6
				8 procs.	**115397**	114806.8	0.0198	3860.2	114161	113312.2	0.0303	253.6
OR10x500-29	500	10	307014	2 procs.	303476	303422.4	0.0115	258.2	**305051**	303728.2	0.0064	274.4
				4 procs.	302425	302196.2	0.0149	606.0	303690	303179.0	0.0108	215.2
				8 procs.	303476	303175.2	0.0115	1886.6	303469	303322.6	0.0115	187.2
OR30x500-00	500	30	115868	2 procs.	112863	111845.0	0.0259	729.2	112620	111334.0	0.0280	380.2
				4 procs.	113453	112673.2	0.0208	1945.0	111340	110791.6	0.0391	319.4
				8 procs.	**113545**	112836.6	0.0200	5097.0	110133	109422.2	0.0495	141.0
OR30x500-29	500	30	300460	2 procs.	298450	297875.8	0.0067	189.4	298211	297746.6	0.0075	181.8
				4 procs.	298193	297565.0	0.0075	487.0	298683	297912.2	0.0059	146.4
				8 procs.	**298701**	298013.6	0.0059	1230.2	297681	297614.4	0.0092	97.6