A Skeleton for the Tabu Search Metaheuristic with Applications to Problems in Software Engineering[*] (Extended Abstract)

Maria J. Blesa Fatos Xhafa

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Campus Nord C6
E-08034, Barcelona, Spain

E-mail: {mjblesa,fatos}@lsi.upc.es

Abstract

We present a C++ implementation of a skeleton for Tabu Search method. Tabu Search method is a well-known meta-heuristic that has proved successful for sub-optimally solving hard combinatorial optimization problems. Recently there is an increasing interest in the application of meta-heuristics, such as Tabu Search, to problems in software engineering. Tabu Search method has already been implemented for a large number of optimization problems, to the best of our knowledge they are all *ad hoc* implementations. We propose a generic C++ implementation based on a skeleton design for the method. This implementation offers, among others, the possibility for the user to instantiate the Tabu Search method for any problem with little efforts and basic knowledge of C++ language. The implementation provides both robustness and re-usability properties due to Object Oriented Paradigm. We show how to model the project management scheduling as an optimization problem and instantiate the skeleton of Tabu Search to solve it.

Keywords: *Meta-heuristics, Tabu Search, Generic Programming, Software Engineering.*

1 Introduction

Meta-heuristics, such as Tabu Search, Simulated Annealing and Genetic Algorithms, have extensively been used for approximately solving optimization problems arising from different areas of theory and practice (combinatorial optimization, graph theory, economics, engineering, etc.). Recently there is an increasing interest in applying meta-heuristic methods for solving software engineering problems. Clarke et al. [CHHJ00] (see also [JES98, TCM98]) showed how meta-heuristics can be applied to problems arising in software engineering, such as software testing, requirements phasing, systems integration, etc. They also suggest a list of problems candidate for such application including the project management scheduling.

One of the well-known meta-heuristics is the Tabu Search (TS). TS was introduced by Glover [Glo77] (for its current form see [Glo86]) and has been successfully applied to many problems such as scheduling problems (e.g. [LBG91, Wid91, DT86, PR95]), graph problems (e.g. k-Cardinality Tree [JL97]), resource allocations (e.g. [SK90, **?**]), layout problems (e.g. [FW74, KP78]), to name a few. TS is a meta-heuristic that means it consists of a main heuristic and several internal heuristics whose implementation depends on the problem at hand. This makes the Tabu Search a quite flexible method since usually different implementations of the internal

[*]This research was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and the CICYT project TIC1999-0754-C03 (MALLBA).

heuristics are possible. Given its wide applicability, TS has now become an established optimization approach that is rapidly spreading to many new fields, such as resource management, human factors engineering, process design, logistics, and technology planning. After a careful and almost exhaustive revision of existing implementations, we have observed that all of them are *ad hoc* and quite dependable on the problem at hand. This approach has, at least, two drawbacks. First, one has to implement the method from scratch for any problem of interest, and, second, it is difficult to introduce even small changes in the code since it would require the modification of most of the implementation. We remark again that different implementations of the tabu search for the same problem can be generated by varying the definition and structure of several underlying entities of the search, i.e. internal heuristics. So, it would be quite interesting to generate new implementations for a problem from existing ones with as many few changes as possible, at the aim of obtaining a better implementation.

In this work, our motivation was to design and implement a skeleton for TS based on Generic Programming Paradigm. The main advantages of this implementation are: (a) it allows the user to instantiate any problem of interest with little efforts (by simply describing problem-dependent features), (b) it is flexible enough so different implementations for the same problem can be generated by changing the implementation of the internal heuristics of the method, and (c) it is easy to use even by users not familiarized with Tabu Search from areas such as biology, economics, software engineering etc. These properties of our implementation are assured by providing a generic implementation of the entities of Tabu Search that do not depend on the problem (e.g. the main method) and a fixed but generic interface for the rest of the entities of the method that are problem-dependent. Therefore, in order to instantiate the skeleton for a given problem the user has to just *fill in*, i.e. complete, the implementation of interfaces for the problem-dependent features. So Software Engineering problems can be reformulated as search problems and solved through our Tabu Search skeleton only by defining some problem-dependent features.

Why generic programming? The ingredients of the Tabu Search method are the same for any optimization problem to which one would like to apply the method. What makes different the implementation of Tabu Search for different problems is the implementation of internal heuristics but the *engine* of the Tabu Search can be made generic enough so as to assume that it doesn't depend on the problem. As we will see later, at this point we had to abstract from a large number of implementation in order to come up with a generic form of the main method. It is, therefore, quite interesting to have a *generic program* or a kind of *template* for Tabu Search from which one could derive instantiations for any problem of interest. Many authors have pointed out that TS may be viewed as an engineering design approach. In this spirit, we dealt with the design and implementation issues of Tabu Search from a generic programming paradigm. The skeleton was, then, achieved by a careful design identifying the common entities of the TS method.

2 Overview on Tabu Search

Tabu Search belongs to the family of local search algorithms but here the search is done in a *guided* way, by maintaining historical information on exploration process, in order to overcome the local optima. Roughly speaking, the method starts from an initial solution and jumps from one solution to another one in the solution space but tries to avoid cycling by forbidding or penalizing moves which take the solution, in the next iteration, to solutions previously visited (called *tabu*). To this aim, TS keeps a *tabu list* which is historical in nature and constitutes the Tabu Search memory. The role of the memory can change as the algorithm proceeds. At initialization the goal is to make a coarse examination of the solution space and further on the search is focused to produce local optima solutions in a process of *intensification* or make a *diversification* in order to explore new regions of the solution space.

2.1 Identifying the Main Entities of Tabu Search

Here we identify the basic entities participating in the Tabu Search method. In order to abstract these entities and functionalities we did a very careful review of different known *ad hoc* implementations for the method in the literature. As we will see, a right abstraction will be easily translated into a programming language using object oriented and generic programming paradigm. This will allow us to obtain a generic and problem-independent skeleton for the Tabu Search method.

Problem. Represents an instance of the problem to be solved.

Solution. Represents a feasible solution to the problem. The acceptability criteria for TS is to find a feasible solution of cost as close as possible to the optimum cost.

Neighborhood. The set of all possible solutions that are reached from a given solution in a single step (called *move*) is referred to as its neighborhood. TS moves from one solution to the "best" solution among all (or part of) possible solutions in its neighborhood. This choice is crucial to the whole process. Once a solution is visited it is considered tabu for some time. A solution in the neighborhood that is marked tabu will not be considered, so cycling, i.e. falling into already visited solutions, is (partially) avoided.

Move. A transition from a feasible solution to another one is called *move*. Typically, as in other local search methods, a move performs some local perturbation over the solution it is applied to. A move may be described by one or more attributes. Considering the number of attributes representing a move we may distinguish single-attribute moves and multi-attribute moves. The reason behind this is that, in general, solutions are extremely impractical to keep track of the exploration process and therefore it is better to describe the exploration process in terms of moves. Moves are given the *tabu status* if they lead to previously visited solutions. However TS establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy such a criteria.

Tabu list. Applying a move to a given solution may result in a "better" or "worst" solution. Without additional control, however, a locally optimal solution can be re-visited immediately after moving to a neighbor, or in a future stage of the search process. To prevent the search from cycling between the same solutions, TS uses a short term memory –the so-called tabu list– to the aim of representing the trajectory of solutions considered. The goal is to permit "good" moves in each iteration without re-visiting solutions already encountered. The tabu list management is a key point to the TS procedure.

Intensification and Diversification. While exploring a region of solution space it seems reasonable to *intensify* the search if we had evidences that such a region may contain good solutions. To this aim, TS incorporates an intensification procedure. During the exploration process the method may get stuck in a region where no better solutions are found. In such a case, the TS method launches the *diversification* procedure to spread out the search in another region. For the sake of genericity we have split of the diversification into *soft diversification* when we move to a region close to the current one and *strong diversification* when we move to a completely different region.

Main procedure. There is no a standard main procedure for TS in the literature. We checked out several existing TS programs and observed that the main procedure looks different in different implementations of TS since the authors adapt it to the problem at hand. Since we wanted a skeleton for TS such that any problem could *fit in*, we had to deal with the design process of a component. This component was going to be the principle engine of any program for TS obtained by instantiating the skeleton. In such a design process we had to abstract from a large number of different implementations for TS and we came up with the component, called Solver.

3 Implementation of the Skeleton for Tabu Search Method

We have designed and implemented a generic skeleton for the TS method. The main entities of the method mentioned in Section 2.1 have been translated into either C++ classes or methods, according to their logical definition in the context and domain of the TS method. Some of these entities have directly become C++ classes (e.g. problem, solution and move) while others have been introduced into classes as methods (e.g. intensification and diversification). The basic idea behind the skeleton is to allow the user to instantiate any optimization problem of interest by only defining the problem-dependent features. Elements related to the inner algorithmic functionality of the method itself are hidden to the user.

The classes forming the skeleton are groupped according to their *"availability"*. The classes implementing inner functionalities of the method (e.g. the main procedure) are completely provided by the skeleton, whereas there are other classes (interfaces) whose implementation is required to be instantiated (completed) by the user. Therefore, the classes forming the skeleton are classified into two groups: *provided* and *required* classes.

Provided Classes. They implement the TS method itself and the rest of inner functionalities. Actually there are only two *provided classes* in the skeleton: the class Solver and the class Setup. The class Solver implements the main procedure and maintains the state of the exploration. The class Setup contains the setup parameters needed to run the method (e.g. number of independent runs to perform, number of iterations per independent run, etc.). The user can consult the state of the search and also inquire other information related to the exploration process. To this end the skeleton offers a transparent interface, i.e. definition, of the provided part to the user. In a certain sense, the *provided classes* can be seen as a "private" part of the skeleton.

Required Classes. They represent the rest of the entities and functionalities involved in the TS method whose representation/implementation depends on the problem being solved. The requirements needed over these entities also depend on the problem. We have been able to abstract the necessities of each entity but the way they are carried out when solving a problem depends strongly on the problem itself. This allowed us to define C++ classes with a fixed interface but no implementation, so that the expected interaction is completely fixed and defined. The class Solver can use the required classes in a "blind and generic way" (i.e. as black boxes) when implementing the TS method but they need a concrete implementation when instantiating a concrete problem.

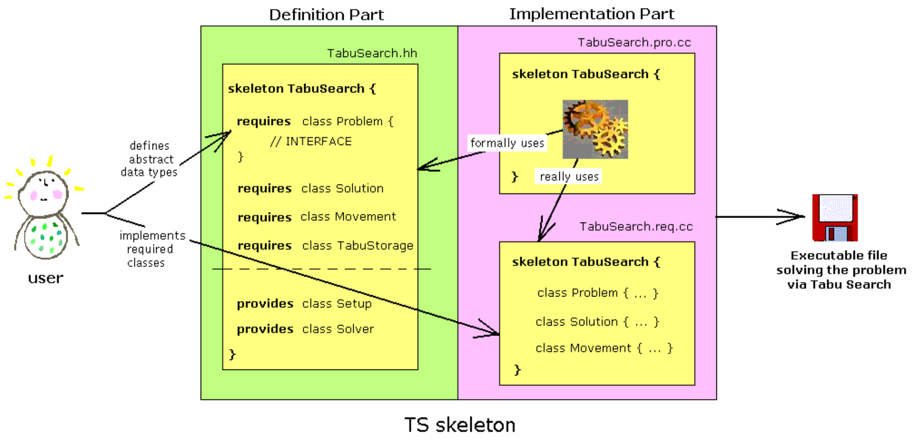

Figure 1: File composition of the TS-skeleton.

Instantiating the skeleton to solve a concrete problem (e.g. a scheduling problem) is the process of completing the requirements of the classes labelled as *required* with the features of a concrete problem at hand. More precisely, to complete a *required class* means: (a) to introduce data types for representing the entities and, (b) to implement the methods of the class according to the chosen data types. We have separated the C++ classes of the skeleton in three parts/files: (1) the interface of the classes (file `TabuSearch.hh`); (2) the implementation of the *provided classes* (file `TabuSearch.pro.cc`) and, (3) the implementation of the *required classes* (file `TabuSearch.req.cc`). Figure 1 represents the relation among required and provided classes and their file organization.

Following we show, by example, how the entities and concepts abstracted in Section 2.1 have been translated into classes and methods (for more details the reader is referred to [BX00]). We will specially focus on describing the provided class `Solver` and the required class `Solution`.

The provided class `Solver` It represents the main procedure of TS method and all the internal features related to the search. Intensification and diversification procedures have been introduced into the exploration process in order to construct a generic standardized algorithm for the TS method.The user can decide if he wants to apply them and, if so, which will be their effect over the neighborhood-based search (see class `Solution` below).

The class `Solver` also collects information about the state of the search being performed. The state basically consists of information about the best solution found so far and those attributes describing the current point of the search process. The interface for class `Solver` now follows:

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

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

        // Global state
       virtual int        independent_run () const;
       virtual float      time_spent () const;
       virtual Solution   best_solution () const;
       virtual int        independent_run_best_found () const;
       virtual int        iteration_best_found () const;
       virtual float      time_best_found () const;
       virtual double     best_cost () const;
       virtual double     worst_cost () const;
       ...
 };
```

The required class `Solution` Represents feasible solutions to the stated problem. Before giving its interface we briefly describe the methods of this class. TS starts exploration from an initial solution generated by some other procedure (typically random or greedy). The initial solution in the TS skeleton is obtained by the method `set_initial()`. Any solution has an associated cost or benefit. The method in charge of calculating this cost in the class is named `fitness()`.

A reachable solution in the neighborhood can be described in terms of the source solution and the move that leads to that neighbor solution after being applied to source solution. The method `apply()` transforms a solution into a neighbor solution by applying a movement. TS will choose the "best" solution in the neighborhood (actually the best move that leads to it) to be the next solution to continue the exploration. Moves in TS are given the *tabu status* if they lead to previously visited solutions. This tends to avoid cycling. Tabu Search also establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy such a criteria. The method `aspiration()` checks this criteria over the current solution with relation to a given movement.

TS incorporates an intensification procedure to intensify the search if it had evidences that the region being explored may contain good solutions. The intensification is done by rewarding solutions having features in common with the current solution, and then solutions that are far from the current solution are penalized indirectly. The method `reward()` has to describe how the current solution (actually the items forming the solution) are rewarded. This method will be invoked just before the intensification starts. Similarly, the methods `penalize()` and `escape()` allow diversification of the search (see Section 2.1).

Resuming, the main part of class `Solution` interface is the following:

```
requires class Solution {
   public:
       Solution (const Problem&  pbm);
       ~Solution();
```

```
        void   set_initial ();
        double fitness () const;
        double delta (const Movement& move) const;
        void   apply ( const Movement& move );
        bool   aspiration (const Movement& move, const TabuStorage& tstore, const Solver& solver) const;
        void   reward ();
        void   penalize ();
        void   escape ();
        ...
};
```

4 Application to Project Management Scheduling

The skeleton for Tabu Search method can be applied to any optimization problem for which we dispose a Tabu Search algorithm, i.e. for which we can specify the Tabu Search entities. In particular, we can apply the skeleton for Tabu Search also to problems arising in software engineering. Clearly, first, we must formulate such problems as optimization problems, second, specify the Tabu Search entities and finally instantiate the skeleton for Tabu Search yielding to an implementation of Tabu Search for the problem at hand. In this section we briefly show these steps for Project Management Scheduling in a general setting (see e.g. [Som96]).

4.1 Project Scheduling as Optimization Problem

A Project Schedule is the description of the Software Process Development. We select an appropriate process model and we identify the (software engineering) tasks that have to be performed. Clearly, each task is associated a duration time (an estimation for the completion time of the task) and there are dependencies between different tasks. Associated to a project are also different resources (e.g. number of people, available machines etc.) needed for the completion of the project and we may assume that resources can be requested by different tasks (see the example given in Figure 2).

Task	Duration	Res Req (1)	Res Req (2)	Res Req (3)	Res Req (4)
1	0	0	0	0	0
2	4	3	9	8	0
3	7	0	0	2	0
4	2	0	0	4	0
5	1	6	0	0	0
6	10	3	0	10	0
7	1	0	1	0	7
8	6	7	0	3	0
9	9	7	10	2	8
10	1	2	0	1	8
11	1	0	3	0	0
12	0	0	0	0	0

Figure 2: Tasks, Duration and Resource Usage

The key point to the Project Management is to assure an optimal completion of the tasks under the precedence constraints and resource constraints. Clearly, this problem is an optimization problem in which the objective is to find a schedule of the activities such that minimizes the overall completion time.

A close observation to the problem shows that this problem belongs to the family of scheduling problem known in optimization theory as Resource Constraint Scheduling Problem (RCSP). We have instantiated the Tabu Search skeleton for the last problem from which we can solve instances of Project Scheduling Problem as well.

4.2 Instantiating the Skeleton

As we explained in Section 3, in order to instantiate the skeleton for the RCPS problem the user should complete the implementation of the interfaces in `TabuSearch.req.cc` such as `Problem`, `Solution`, `Movement` etc. Due to lack of space we show the instantiation process by example, concretely, let us see how would the user complete the implementation of class `Problem` and `Solution`.

Problem representation. The RCPSP can be represented with six attributes: (1) the number of tasks to be scheduled, (2) the number of resources, (3) a matrix representing the precedence relations, (4) an array containing the duration of each task, (5) an array containing the maximum availability of each resource, and (6) a matrix indicating, for each task, how many resources uses of each type. Clearly, we can *fill in* the data representation of class `Problem` by simply adding

```
int _nb_tasks;
int _nb_resources;

array2<int> _precedences;
array<int>  _durations;
array<int>  _resources;
array2<int> _usage;
```

to the private part of this class.

Solution representation. A solution for the scheduling problem is a sequence indicating the order in which tasks should be executed. Tasks are processed from left to right, and one task would be scheduled to start as soon as possible according to its precedence constraints w.r.t. the task that are not scheduled yet and w.r.t. the resource usage of the tasks that are being executed at the same time (resource constraints).

Solutions can be represented with two attributes: (1) an array of integers where each integer represents a task and their index in the array indicates the scheduling order, and (2) a reference to the associated problem, because the problem contains all the information related to precedences, durations, etc.

```
array<int> _schedule;
Problem    _problem;
```

As for the methods of this class, we mention here how is implemented the method `set_initial()`. An initial solution is obtained by a backward deep-first traversal of the precedence graph (without repeating tasks). This is a trivial way to deal with precedence constraints. Resource usage do not influence this procedure due to the way we interpret a solution. In this initial solution, a task is scheduled immediately after some of its precedence tasks.

4.3 Running the program

Once the instantiation is completed, the user may run it with a program like this:

```
#include "TabuSearch.hh"

int Main (int argc, char** argv)
{
    using skeleton TabuSearch;

    Problem problem;              // Read the problem instance.
    cin >> problem;
    Setup setup;                  // Read the setup parameters.
    cin >> setup;

    Solver_Seq solver(problem,setup);  // Run the Tabu Search method.
   solver.run();

    cout << solver.best_solution() << endl;   // Report best solution found.
    cout << solver.best_cost()     << endl;

    return 0;
}
```

5 Conclusions and Future Work

We have designed and implemented a skeleton for Tabu Search Method based on generic programming and object oriented paradigms. The skeleton can be instantiated for any optimization problem, in particular to those problems arising in software engineering that can be formulated as optimization problems. The skeleton offers several advantages to the user such as a standardized form of Tabu Search, less efforts at implementing the Tabu Search as compared to ad hoc implementation and time savings. Due to these properties we believe that the skeleton will be useful not only to users from optimization but also from other different areas where optimization problems arise.

We plan to instantiate the skeleton for other problems in software engineering reported in [CHHJ00]

References

[BX00] M.J. Blesa and F. Xhafa. A C++ Implementation of a Skeleton for Tabu Search Method. Technical Report LSI-00-47-R, Dept. de LSI, UPC, 2000.

[CHHJ00] J. Clarke, M. Harman, R. Hierons, and B. Jones. The Application of Metaheuristic Search Techniques to Problems in Software Engineering. Technical Report SEMINAL-TR-01-2000, SEMINAL Network, 2000. http://www.discbrunel.org.uk/seminal.

[DT86] M. Dell'Amico and M. Trubian. Applying Tabu Search to the Job-Shop Scheduling Problem. *Ann. of Op. Res.*, 41:231–252, 1986.

[FW74] R.L. Francis and J.A. White. *Facility Layout and Location.* Prentice-Hall, 1974.

[Glo77] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[Glo86] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Op. Res.*, 5:533–549, 1986.

[JES98] B. Jones, D. Eyres, and H. Sthamer. A Strategy for Using Genetic Algorithms to Automate Branch and Fault-Based Testing. *The Computer Journal*, 41:98–107, 1998.

[JL97] K. Jörnsten and A. Løkketangen. Tabu Search for Weighted k-Cardinality Trees. *Asia-Pacific J. of Op. Res.*, 14(2):9–26, 1997.

[KP78] J. Krarup and P.M. Pruzan. Computer-aided Layout Design. *Math. Prog. Study*, 9:75–94, 1978.

[LBG91] M. Laguna, J.W. Barnes, and F. Glover. Tabu Search Methodology for a Single Machine Scheduling problem. *J. of Int. Manufacturing*, 2:63–74, 1991.

[PR95] S. Porto and C. Ribeiro. A Tabu Search Approach to Task Scheduling on Heterogeneous Processor under Precedence Constraints. *Int. J. of High-Speed Comp.*, 7, 1995.

[SK90] J. Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *ORSA J. on Comp.*, 2(1):33–45, 1990.

[Som96] I. Somerville. *Software Engineering.* Addison Wesley, 1996.

[TCM98] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, 1998.

[Wid91] M. Widmer. The Job-shop Scheduling with Tooling Constraints: A Tabu Search Approach. *J. Op. Res.*, 42:75–82, 1991.

A Experimental Results

We have tested the instantiation TS skeleton for the Resource-Constrained Project Scheduling Problem (RCPSP) with instances from the literature. The table below shows the results obtained for some small instances obtained from the Institut fü Wirtschaftstheorie und Operations Research[1] at the Universität Karlsruhe. By know we have tested a simpler version of these instances by not considering the weights of the arcs in the graph of precedences.

Instance	Nb. Tasks	Nb. Resources	Best Solution (Fitness) Obtained	Time Needed (secs)	Total Execution Time (secs)
TESTSETUBO.psp1.sch	10	5	66	0.06	2754.25
TESTSETUBO.psp2.sch	10	5	38	0.19	1251.03
TESTSETUBO.psp3.sch	10	5	38	213.14	1496.76
TESTSETUBO.psp4.sch	10	5	40	0.08	2165.11
TESTSETUBO.psp5.sch	10	5	42	0.91	1400.07
TESTSETUBO.psp16.sch	10	5	28	0.05	737.11
TESTSETUBO.psp17.sch	10	5	56	0.58	808.37
TESTSETUBO.psp18.sch	10	5	44	117.24	1455.4
TESTSETUBO.psp19.sch	10	5	32	1.41	907.15
TESTSETUBO.psp20.sch	10	5	42	0.04	184.05
TESTSETUBO.psp21.sch	10	5	29	0.17	372.37

All the executions performed share the same setup configuration:

· Number of Independent Runs: 10
· Number of Iterations per IR: 1000
· Use Delta Function: NO
· Tabu Size: 10
· Mininum Tabu Status: 1 iteration
· Maximum Tabu Status: 1 iteration
· Maximum Number of Repetitions to Block the Search: 10
· Diversify during: 50 iterations
· Intensify during: 50 iterations

We want to test in the near future other interesting benchmarks. One is the *Online Resources on Scheduling*[2]. This site informs about the *Research Group Resource Constrained Project Scheduling* (see www.wior.uni-karlsruhe.de/rcpsp/) at the Institut fü Wirtschaftstheorie und Operations Research.

Another important site is the Library PSPLIB[3] at the Institute der Wirtschafts und Sozialwissenschaftlichen Fakultät der Christian Albrechts Universität zu Kiel. This library contains different problem sets for various types of resource constrained project scheduling problems as well as optimal and heuristic solutions. The instances have been generated by the standard project generator ProGen. The library itself, i.e. the types of models represented, details of the generation of the problems, the experimental design for generating the problems, problem parameters etc, can be found in the following paper:

Kolisch, R. and A. Sprecher (1996): *PSPLIB - A project scheduling library*,
European Journal of Operational Research, Vol. 96, pp. 205–216.

The original working paper can be downloaded via the home-page or ftp[4].

We plan to test our instantiation with some of these benchmarks and publish the obtained results at www.lsi.upc.es/∼mjblesa/TSExperiments/scheduling.html.

[1]www.wior.uni-karlsruhe.de/RCPSPmax/progenmax/rcpspmax.html
[2]www.ie.bilkent.edu.tr/ lors/lors/genc3.html
[3]www.bwl.uni-kiel.de/Prod/psplib/library.html
[4]www.bwl.uni-kiel.de/bwlinstitute/Prod/mab/kolisch.html
ftp.bwl.uni-kiel.de/pub/operations-research/wp396.ps