A C++ Implementation of a Skeleton for Tabu Search Method*

M. Josep Blesa and Fatos Xhafa

Departament de LSI
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona, 1-3
08034-Barcelona, Spain.
Phone: + 34 93 401 70 03
Fax: + 34 93 401 70 14
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. This method has been already 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. We have instantiated the skeleton on three different problems, namely the Quadratic Assignment, Minimum k-Cardinality Tree and 0-1 Multi-dimensional Knapsack problems so as to evidence the claimed properties of the skeleton.

Keywords: *Generic Programming, Reuse, Components, Optimization, Tabu Search.*

1 Introduction

Tabu Search (TS) is a meta-heuristic method designed for the solution of hard optimization problems. TS was first introduced by Glover [8] (for its current form see [9]) and has been used to sub-optimally solve a wide range of important optimization problems from both theory and practice such as scheduling problems (e.g. [14, 23, 4, 18]), graph problems (e.g. k-Cardinality Tree [11]), resource allocations (e.g. [19, 21]), layout problems (e.g. [20, 7, 13]),

*This research was partially supported by the ALCOM-FT (IST-99-14186) and CICYT project TIC1999-0754-C03.

to name a few. 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. Interestingly, TS is also used as an *intelligent agent* by other procedures such as hybrid algorithms [6] and memetic algorithms [16, 17].

Tabu Search belongs to the family of local search algorithms but here the search is done in a *guided* way 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 forms 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.

There are several implementations of TS in the literature for different problems (see, e.g. [1, 22, 11]). All of them, however, 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 the 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 here that different implementations for the same problem can be generated by varying the definition and structure of the tabu list (for example by deciding how tabu attributes are determined), the size of the candidate list, intensification and diversification procedures etc., so it would be quite interesting to generate new implementations for a problem from existing ones with as many few changes as possible.

Many authors have pointed out that TS may be viewed as an engineering design approach. In spirit of this idea, we provide a generic skeleton for TS implemented in C++. Our motivation for this was twofold: first, to enable the user to instantiate the TS for any problem with little efforts by re-using several components, second, to give flexibility at implementing the method. These properties are achieved by a careful design of the skeleton identifying the common entities of the TS method and, of course, due to object oriented technology. We have instantiated the skeleton for three different problems, namely the Quadratic Assignment, k-Cardinality Tree and 0-1 Multi-dimensional Knapsack so as to evidence the claimed properties of the skeleton. We have observed the easy of use of the skeleton, the flexibility, robustness as well as a considerable savings in time. It is generally believed that the success of the TS depends on the careful implementation for the problem at hand, or, in other terms, an efficient implementation should incorporate as much knowledge of the problem as possible. In spite of this we show that even though being generic, our implementation achieves comparable results to those of other specific implementations and in some cases better ones.

The paper is organized as follows. In Section 2 we overview the main ingredients of the TS method so as to identify the entities of our skeleton. Then, in Section 3 we give the design and the implementation of the skeleton and in Section 4 we show how to instantiate our skeleton for a given problem and show it concretely for Min k-Cardinality Tree. We make a comparison of our results with known ones and give some conclusions in Section 5.

We include also extended experimental results in Appendix A for Quadratic Assignment and k-Cardinality Tree Problems.

2 Basic Ingredients of Tabu Search Method

In this section we review the basic ingredients of the tabu search method so as to identify the main entities of the skeleton and exemplify them through the k-Cardinality Tree –an important problem to both graph theory and oil field leasing from industry.

Problem. We are given an optimization problem consisting of: (a) a set of instances $\mathcal{I}$; (b) to a given instance $x \in \mathcal{I}$, there corresponds a set of feasible solutions $\mathcal{S}(x)$ (hereafter denoted by $\mathcal{S}$); (c) a cost function $f : \mathcal{S} \to \mathcal{R}$ associating a cost $f(s)$ to solutions $s \in \mathcal{S}$. We want to find a solution $s^* \in \mathcal{S}$ with respect to some criteria (minimization or maximization).

In terms of k-Cardinality Tree, an instance is a weighted undirected graph G and a natural k –the cardinality of the tree. The set of feasible solutions consists of all the subgraphs of G that are trees of k edges. The cost of a solution is the weight of the tree, i.e. the sum of the weights of the edges. We seek for a k-tree of minimum weight.

Solution. Tabu Search seeks for a feasible solution, as defined previously, but the acceptability criteria for TS is to find a solution $s' \in \mathcal{S}$ of cost $f(s')$ as close as possible to the optimum cost $f(s^*)$. Clearly, TS is an heuristic procedure (there is no guarantee that the optimal solution will be found.)

For k-Cardinality Tree the solution s' (the k-tree) is obtained by fixing a stopping criteria, e.g. the number of iterations to be done by the algorithm.

Neighborhood. Given a solution s, the set of all possible solutions that are reached from s in a single step (called move), is referred to as the neighborhood of s, denoted by $\mathcal{N}(s)$. TS moves from s to $s' \in \mathcal{N}(s)$ which is *the best* among all (or part of) possible solutions in $\mathcal{N}(s)$. The choice of the next solution s' is crucial to the whole process. To carry it out recently visited solutions (marked as *tabu*) should be avoided. So, once a solution is visited it is considered tabu, hence $\mathcal{N}(s)$ will be changing along the exploration.

For k-Cardinality Tree, the neighborhood of a solution T, i.e. a tree, consists of all the trees that could be obtained from T by swapping an edge in T with another not in T while maintaining the tree condition.

Move. A transition from a feasible solution to another feasible solution is called *move*. Notice that a move may be described by one or more attributes, and these attributes can become the foundation for creating an attribute based memory. 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 $\mathcal{S}$ in terms of moves. Moves can be given the *tabu status* if they lead to previously visited solutions, but TS establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy such a criteria.

For k-Cardinality Tree, we have multi-attribute moves, namely a move consists of the edge to be removed, the edge to be added, the type of the move (it can be dynamic or static) and the tabu life (the number of iterations the move remains tabu). A move is considered tabu if it tries to add an edge that is recently dropped from the tree or drops an edge that is recently added to the tree. But, in spite of being tabu, it can be accepted if it improves the cost when applied to the current solution (the k-tree).

Tabu list and efficient use of memory. Applying a move to a given solution may result in a improvement or a deterioration of the objective function value. Without additional control, however, such a process can cause a locally optimal solution to be re-visited immediately after moving to a neighbor, or in a future stage of the search process, respectively. 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 encountered. Based on certain restrictions the tabu list implicitly keeps track of moves by recording their attributes. These attributes will be forbidden from being embodied in moves selected in at least one subsequent iteration because their inclusion might lead back to a previously visited solution. The goal is to permit "good" moves in each iteration without re-visiting solutions already encountered. The key point to the TS procedure lies in the tabu list management. Basically it concerns updating the tabu list, i.e., deciding on how many and which moves have to be set tabu within any iteration of the search.

For the k-Cardinality Tree, we can maintain two lists, one for the dropped edges and another one for the added ones.

Intensification. 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 that consists in rewarding solutions having features in common with the current solution and penalizing solutions that are far from the current solution. Usually, the intensification is carried out over a few iterations.

For the k-Cardinality Tree, we can intensify by rewarding those edges of minimum weight that have occurred in the local optima.

Diversification. In the exploration process, the method may get stuck in a region where no good solutions are found even a reasonable number of iterations is done. In such a case, the TS method launches the *diversification* procedure to spread out the search in another region. The diversification may be *softly* meaning that a region close to the current one is going to be explored, or *strongly* when a completely different region is considered.

For the k-Cardinality Tree, we may diversify softly by penalizing the edges that have occurred in the local optima while we may apply an *escape* for the strong diversification.

Main procedure. There is no a standard main procedure for TS in the literature. We have checked out several existing TS programs and have observed that the main procedure looks different in different implementations of TS since the authors adapt it to the problem at hand. Since we want 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. For simplicity, we give the diagram of Figure 1 where we can easily observe the main functionalities of such a component (see Section 3 for its interface.)

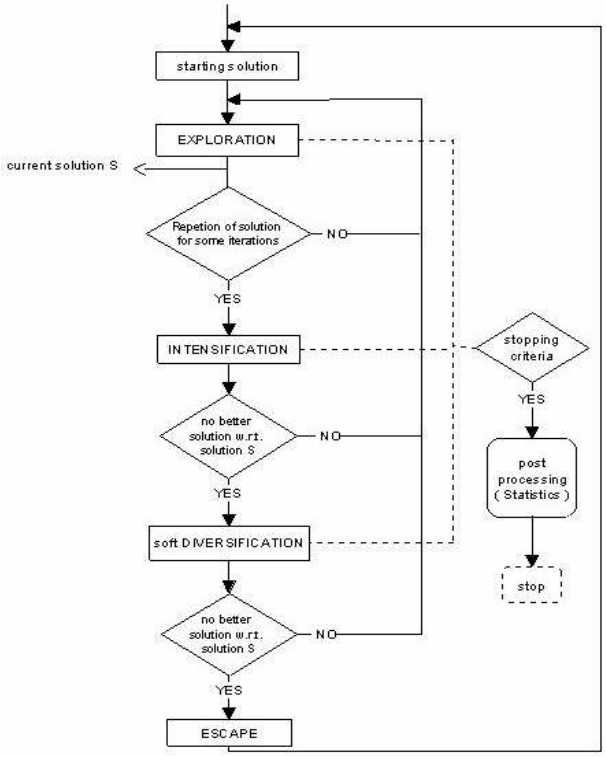

Figure 1: Main procedure of Tabu Search method.

3 C++ Implementation of a Skeleton for Tabu Search

In this section we show the main points of the design and implementation in C++ of our skeleton[1] for Tabu Search.

In order to design the skeleton we did a careful revision of the TS method and of a large number of implementations for different problems known in the literature. If we look closely to the basic ingredients of TS, as given in the previous section, we see that: (a) there are entities, such as Problem, Solution, that are completely problem dependent; (b) there are entities, such as tabu list, which are generic but the user may want to benefit himself from the structure of the problem at hand to implement such a list; (c) there are entities, such as the Solver (see main procedure), Statistics, that do not dependent on the problem at hand, in that, they can be implemented in a generic way. Hence our skeleton have two distinguished parts: a *"public"* part consisting of classes to be filled in by the user who will instantiate the skeleton for a given problem (these classes are gathered in `TabuSearch.hh` and `TabuSearch.cc`) and a *"private"* part consisting of several generic classes (or *components*) already *filled in*, i.e. implemented (gathered in `TabuSearch.imp.hh` and `TabuSearch.imp.cc`). Following we give the interface of two classes for a flavor of how the classes from the two parts look like.

Solution: Required abstract class, extends Serializable[2]. Corresponds to the definition of a

[1] The Longman dictionary defines a skeleton as: *Something forming a structure on which more is built or added.*

[2] Public abstract class that takes care of the I/O.

solution of a problem instance. The user who instantiates the skeleton must provide a complete definition of this class. This definition has to implement:

Solution(): Constructor.

feasible(): Indicates whether the solution is feasible.

fitness(): Measures the cost function on the solution.

initial_solution(): Constructs an initial solution.

escape(): Permits to diversify.

delta(): Permits an efficient computation of fitness.

aspiration(): Indicates whether the tabu solution has aspiration.

apply(): Applies a given move to the solution.

unapply(): Unapplies a given move to the solution.

penalize(): Permits penalizing some attributes of the solution.

unpenalize(): Removes the penalties to a given solution.

reward(): Permits rewarding some attributes of the solution.

unreward(): Removes the rewards to a given solution.

Solver: Offered implemented class –a component. This class provides the functionalities to operate the Tabu Search heuristic, in particular, it offers the following functionalities: (a) Initialization of the Solver with a problem and a configuration; (b) Execution: calls for the execution of the algorithm implemented in the skeleton; (c) Post-execution information: the best solution ever seen, the best cost ever seen, the worse cost ever seen and other statistical information. The interface of Solver is as follows.

Attributes:

- problem // Instance problem.
- params // Set up parameters.
- bsolution // Best solution.
- state // State of the algorithm.
- statistics // Post-execution information.
- structure // Structure for managing the tabu moves.
- bcost, wcost // Best and worst cost, respectively.

Methods:

- Solver () //Constructor.
- run () // Runs the main procedure of tabu search.
- get_state (); get_statistics ()
- best_solution ();
- best_cost (); worst_cost ()

Our skeleton offers also the possibility for the user to extend several classes of the skeleton (which can be viewed as *extendable components*) in order to give more flexibility to the instantiation of the skeleton. For instance, the user can extend the Statistics class to a UserStatistics that permits the user to incorporate other statistics specific to the problem at hand; similarly, the user can extend the TabuList –which provides the management of tabu moves and efficiency issues– to UserStructure thus allowing the user to do this management using specific knowledge of the problem at hand. It's worth mentioning here that for both components Statistics and TabuList we dealt with the design process of a component, as we did for the Solver. These components were going to be part of the instantiated TS program. Again, in such a design process we had to abstract from a large number of different implementations for TS.

Notice that we have two levels of re-usability: in the method itself and in the instantiations. The re-usability of the method is due to the standardization of the Tabu Search algorithm. This component is directly reused from the TS skeleton without any effort for the user. The second level refers to the instantiations already done. There exist many shared entities that can be implemented in the same way for many problems, so a unique implementation can be reused for all of them (for example, the k-Cardinality Tree problem and the Traveling Salesman problem are represented by an undirected weighted graph, so in this case, the class representing the problem can be reused). Furthermore, by doing small changes in an instantiation, and reusing the rest of it, different implementations of the same problem can be obtained. The power of re-usability is due to the Object Oriented strategy used to organize the entities and the careful design of the entities itself.

4 An Instantiation Example: Min k-Cardinality Tree

We have instantiated the TS skeleton for the Quadratic Assignment, k-Cardinality Tree and Knapsack. Here we exemplify the instantiation process for the case of k-Cardinality Tree. The TS algorithm for the problem was first given in [11] and then revisited in [10].

Let us now show how would the user instantiate the skeleton for the k-Cardinality Tree problem. Starting from `TabuSearch.hh` and `TabuSearch.cc` the user obtains `Min_kTree.hh` and `Min_kTree.cc`, respectively. For example, in `TabuSearch.hh` the user finds:

```
class Problem {
 public:
  Problem ();
  Direction direction () const;

  Problem& operator=  (const Problem& pbm);
  bool     operator== (const Problem& pbm) const;
  bool     operator!= (const Problem& pbm) const;

  friend ostream& operator<< (ostream& os, const Problem& pbm);
  friend istream& operator>> (istream& is, Problem& pbm);
  friend opacket& operator<< (opacket& op, const Problem& pbm);
  friend ipacket& operator>> (ipacket& ip, Problem& pbm);
};
```

and instantiates it to Min k-Cardinality Tree as follows:

```
class Problem {
 public:
  int  K;
```

```
  int  nb_nodes;
  UGRAPH<int,double> G;

  Problem ();
  Direction direction () const;

  Problem& operator= (const Problem& pbm);
  bool operator== (const Problem& pbm) const;
  bool operator!= (const Problem& pbm) const;
  friend ostream& operator<< (ostream& os, const Problem& pbm);
  friend istream& operator>> (istream& is, Problem& pbm);
  friend opacket& operator<< (opacket& op, const Problem& pbm);
  friend ipacket& operator>> (ipacket& ip, Problem& pbm);
};
```

As another example, let us see the class Solution.

```
class Solution {

public:

  Solution (const Problem& pbm);

  void   initial_solution();
  void   escape();

  double fitness () const;
  double delta (const Movement& mov) const;
  bool   aspiration (const Movement& mov, const TabuStorage& ts,
         const State& state) const;

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

  void   penalize();
  void   unpenalize();

  void   reward();
  void   unreward();

  Solution& operator=  (const Solution& sol);
  bool      operator== (const Solution& sol) const;
  bool      operator!= (const Solution& sol) const;

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

```
  friend opacket& operator<< (opacket& op, const Solution& sol);
  friend ipacket& operator>> (ipacket& ip, Solution& sol);
 };
```

and obtains

```
class Solution {

 public:

  Problem& problem;

  set<edge> penalized;
  set<edge> rewarded;

  set<edge> kTree;
  set<node> kTreeNodes;

  Solution (const Problem& pbm);
  void   initial_solution ();
  void   escape();
  double fitness () const;
  double delta (const Movement& mov) const;
  bool   aspiration (const Movement& mov, const TabuStorage& ts, const State&
         state) const;
  void   apply ( const Movement& mov );
  void   unapply ( const Movement& mov );
  void   penalize();
  void   unpenalize();
  void   reward();
  void   unreward();
  set<node>  kTree_node_leaves() const;
  set<edge>  kTree_edge_leaves() const;
  list<edge> kTree_sorted_incident_edges() const;
  Solution& operator= (const Solution& sol);
  bool operator== (const Solution& sol) const;
  bool operator!= (const Solution& sol) const;
  friend ostream& operator<< (ostream& os, const Solution& sol);
  friend istream& operator>> (istream& is, Solution& sol);
  friend opacket& operator<< (opacket& op, const Solution& sol);
  friend ipacket& operator>> (ipacket& ip, Solution& sol);
 };
```

Similarly, in `TabuSearch.cc` the user finds, for instance:

```
double Solution::fitness () const
{
	double cost;
	return cost;
}
```

and instantiates it to Min k-Cardinality Tree as follows:

```
double Solution::fitness () const
{
	edge e;
	double cost = 0;
	forall(e,kTree) cost += problem.G[e];
	return cost;
}
```

As another example, consider the method apply of class Solution. In `TabuSearch.hh` the user finds:

```
void Solution::apply (const Movement& mov)
{
}
```

and obtains

```
void Solution::apply (const Movement& mov)
{
  node u = problem.G.source( mov.edgeToRemove );
  node v = problem.G.target( mov.edgeToRemove );
  node x = problem.G.source( mov.edgeToInsert );
  node y = problem.G.target( mov.edgeToInsert );

   if (!(((u==x) && (v==y)) || ((u==y) && (v==x))))
    {
      if (mov.typeOf == DYNAMIC)
      {
       if (kTree_degree( problem.G, kTree, u ) == 1)
                        kTreeNodes.del(u);
         if (kTree_degree( problem.G, kTree, v ) == 1)
                   kTreeNodes.del(v);
       }
       kTree.del( mov.edgeToRemove );
       if (!kTreeNodes.member(x)) kTreeNodes.insert(x);
         if (!kTreeNodes.member(y)) kTreeNodes.insert(y);
```

```
        kTree.insert( mov.edgeToInsert );
    }
}
```

We remark that in order to complete the instantiation process it is required that the user knows the Tabu Search method for his problem and sort of basic knowledge of C++.

5 Comparison Results and Conclusions

In order to evaluate the performance of our skeleton for TS we have run the instantiated programs TSQAP (TS for Quadratic Assignment Problem), TSKTree (TS for k-Cardinality Tree) and TSKnapsack (TS for Knapsack) on several test instances, as explained below. We run the instances on a PC (Processor: AMD K6-2 450 MHz, Main memory: 256 Mb, HD: 6 Gb) running Linux as operating system (Redhat 6.1, kernel 2.2.12-20).

5.1 Experimental Results on QAP

For the QAP, we consider several instances from the QAPLIB [3] (QAPLIB is a standard library for QAP) ranging from $n = 9$ to $n = 256$ locations. We show (part of) the results in Table 1, where instance denotes the name of the QAP instance (the number indicates the size of the instance), best known indicates the best known fitness for the instance, TSQAP column gives the fitness found by our TSQAP within 10 independent runs and 100 iterations per run, gap indicates the % deviation from the best known fitness.

instance	best known	TSQAP	
		fitness	gap
tai9	94622	94622	0%
tai10	135028	135028	0%
tai12	230704	224416	-2.72%
tai15	388214	388250	0.009%
tai35a	2422002	2465270	1.78%
rou12	235528	235852	0.13%
rou15	354210	354210	0%
rou20	725522	730090	0.62%
scr12	31410	31410	0%
scr15	51140	51140	0%
scr20	110030	110058	0.02%

Table 1: Comparison of fitness found by TSQAP with best known fitness.

We can see from Table 1 that our instantiation TSQAP program obtains, for those instances, fitness that are very close to the best known, and improves the best so far fitness for tai12.

We also compare the results obtained by our TSQAP with the results obtained by other heuristics and other implementations of the TS, given in Table 2 in which Re-TS stands for Reactive TS [1], Ro-TS for Robust TS [21], MMAS for Min-Max Ant System and MA denotes memetic algorithm. The numerical values, except for TSQAP column, are reproduced from TABLE IV of [15]. We remark that the results from [15] have been obtained in a platform unknown to us; they did 30 independent runs and fixed before the hand the run time[3] while we did 10 independent runs with 100 iterations per run. More experimental results can be found in Appendix A.1.

instance	best known	Re-TS gap %	Ro-TS gap %	MMAS gap %	MA gap %	TSQAP gap %
nug30	6124	0.589	0.020	0.00	0.004	0.06
kra30a	88900	0.612	0.223	0.119	0.369	2.0
tai80a	13557864	0.482	1.023	0.768	1.106	2.1
tai100a	21125314	0.385	0.909	0.728	1.09	1.82
sko100a	152002	0.397	0.191	0.195	0.096	1.02
tai150b	498896643	1.77	2.85	1.16	0.36	2.4
tai256c	44759294	0.266	0.326	0.067	0.070	0.31

Table 2: Comparison of TSQAP with other methods and implementations.

5.2 Experimental Results on k-Cardinality Tree

For the k-Cardinality Tree problem, to the best of our knowledge, there is no a standard library of benchmarks. There are, however, benchmarks from two interesting sources. The first one is given by Jörsten and Løkketagen [11] which includes executable code for generating instances of the problem as well as executable code for solving the k-Cardinality Tree with TS method. The second source is that of Ehrgott et al. [5] including an extensive list of tests as well executable code consisting of several heuristics for solving the problem.

Jörsten and Løkketagen benchmark. Table 3 resumes the results obtained by our TSKTree instantiation with those of Jörsten and Løkketagen TS algorithm (abbreviated JLTS). We generated several graph instances following the methodology and notations of [11] and run both programs for $k = 20$ for all instances. The instance name gxx-4-yy means graphs of xx vertices, we fixed to 4 the degree of any vertex (as in [11]) and yy indicates the index of the instance (we generated and proved several different instances of the same configuration). The weights of the edges were randomly chosen from $\{1, \ldots, 100\}$.

We observed that our TSKTree instantiation matches the results of JLTS. This evidences the good performance of our TSKTree program obtained by instantiating the Tabu Search skeleton.

[3]We will reproduce the results of Table 2, in a later version, once we will have the configuration used in [15]

instance	#vertices	#edges	Best Fitness of JLTS	Best Fitness of TSKTree
g25-4-01	25	50	219	219
g25-4-04	25	48	620	620
g50-4-01	50	98	466	463
g50-4-03	50	99	580	577
g75-4-02	75	150	307	296
g75-4-03	75	149	421	426
g100-4-03	100	200	412	412
g100-4-04	100	199	442	442
g200-4-04	200	399	310	321
g200-4-05	200	399	380	360
g400-4-03	400	799	302	306
g400-4-04	400	800	315	314
g1000-4-02	1000	1999	321	292
g1000-4-05	1000	2000	284	280

Table 3: Comparison of our TSKTree with JLTS implementation.

Ehrgott et al. benchmark. Ehrgott et al. [5] have implemented several heuristics, other than Tabu Search, for solving the Min k-Cardinality Tree problem. We compare with only two of them, namely the DPT (DynamicPrimTree) and DDP (DynamicDijkstraPath) because these two seem to perform best. We run the two heuristics (from the program of [5]) and our TSKtree on the same instances (graphs of 30 vertices and weights uniformly chosen from $\{1, \ldots, 100\}$). The value of k ranges from 3 up to 28. We resume the results in Table 4, more results are given in Appendix A.2.

instance	DPT	DDP	TSKTree
gr30_10_k5	27	25	25
gr30_10_k10	58	54	54
gr30_10_k15	92	92	92
gr30_10_k20	139	139	139
gr30_10_k25	200	200	200
gr30_20_k3	11	11	11
gr30_20_k8	41	41	40
gr30_20_k13	73	73	73
gr30_20_k18	113	113	113
gr30_20_k23	189	189	189
gr30_20_k28	316	316	316

Table 4: Comparison of our TSKTree with other heuristics.

We observed that for the whole benchmark our TSKTree instantiation matches the results of Ehrgott et al. It should be mentioned, however, that the two heuristics we compare with, find solutions that are very close to the optimum, some of them being optimum. Again this evidences the good performance of our Tabu Search skeleton instantiation.

We have also included results on time efficiency for both benchmarks (see Appendix A.2.)

For the 0-1 Multi-dimensional Knapsack, we are currently testing the TSKnapsack on instances from OR-Library [2].

Acknowledgment

We thank the Barcelona Team of Mallba Project for making possible this research. Special thanks go to Jordi Petit i Silvestre for his support while conducting this research. We are grateful to A. Lokketangen, M. Ehrgott and H. Hamacher for providing us with their benchmarks and executable codes for k-Cardinality Tree. We also thank H. Saïd for the benchmark on Knapsack and P. Moscato for bringing to our attention several references.

References

[1] R. Battiti and G. Tecchiolli. C Code for Reactive Tabu Search. http://rtm.science.unitn.it/~battiti/archive/code/rts_qap/, 1995.

[2] J.E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *J. of Op. Res. Soc.*, 41(11):1069–1072, 1990.

[3] R.E. Burkhard, S.E. Karisch, and F. Rendl. QAPLIB – A Quadratic Assignment Problem Library. http://serv1.imm.dtu.dk/~sk/qaplib/, 1996.

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

[5] M. Ehrgott, J. Freitag, H. Hamacher, and F. Maffioli. Heuristics for the k-Cardinality Tree and Subgraph Problem. *Asia-Pacific J. of Op. Res.*, 14:87–114, 1997.

[6] C. Fleurent and J. Ferland. Genetic Hybrids for the Quadratic Assignment Problem. In P. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems.* Amer. Math. Soc., 1994.

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

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

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

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

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

[12] M. Josep 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.

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

[14] 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.

[15] P. Merz and B. Freisleben. Fitness Landscape Analysis and Memetic Algorithms for the Quadratic Assignment Problem. *IEEE Trans. on Evol. Comp.*, 2000. To appear.

[16] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms, and Martial Arts: Towards Memetic Algorithms. Technical Report No. 790, California Institute of Technology, 1989.

[17] P. Moscato. Memetic Algorithms: A Short Introduction. In M. Dorigo D. Corne and F. Glover, editors, *New Ideas in Optimization.* McGraw-Hill, 1999.

[18] 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.

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

[20] L. Steinberg. The Backboard Wiring Problem : A Placement Algorithm. *SIAM Review*, 3:37–50, 1961.

[21] E. Taillard. Robust Tabu Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[22] E. Taillard. Taboo Search Procedure for Asymmetrical Quadratic Assignment Problems. http://sunst50.einev.ch/prive/pro/taillard/codes.dir/robust_ts_qap.p, 1994.

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

A Extended Experimental Results

Here we give extended experimental results obtained on the benchmarks mentioned through the paper. The complete tables of the experimental results can be found also in the web page http://www.lsi.upc.es/∼mjblesa/TSExperiments.

A.1 Experimental Results on QAP Instances

In Table 5 and Table 6 we give results obtained for QAP instances from the QAPLIB library [3]. Table 5 includes also execution times.

instance	best known	TSQAP		
		obtained	gap%	t/s (10 trials, 1000 it/trial)
bur26d	3821225	3821225	0	360.88
kra30a	88900	90720	2.04	550.39
kra30b	91420	91590	0.18	539.21
nug30	6124	6128	0.06	541.32
rou12	235528	235528	0	19.30
rou15	354210	354210	0	36.55
rou20	725522	726100	0.07	114.09
scr12	31410	31410	0	18.58
scr15	51140	51140	0	40.92
scr20	110030	110030	0	118.90
ste36b	15852	16112	1.64	1221.77
ste36c	8239110	8302920	0.77	1160.54
tai9	94622	94622	0	6.87
tai10	135028	135028	0	9.15
tai12	224416	224416	0	18.49
tai15	388214	388214	0	43.27
tai20	703482	705622	0.30	108.82
tai35a	2422002	2458830	1.52	922.21
tai50	4941410	5052310	2.24	3339.36

Table 5: Comparison of TSQAP with other best known fitness from QAPLIB.

A.2 Experimental Results on Min k-Cardinality Tree Instances

In Table 7 we give results concerning the benchmark of Jörsten and Løkketagen and in Table 8 and Table 9 the ones for (part of) the benchmark of Ehrgott et al. Table 7 and Table 9 include also execution times. We have observed that our TSKtree implementation runs faster than the one of Jörsten and Løkketagen, we remark however that their implementation is done in Windows and includes also a graphical interface which gives an

instance	best known	TSQAP	
		obtained	gap%
esc128	64	64	0
sko100a	152002	153556	1.02
sko100b	153890	155132	0.80
sko100c	147862	150044	1.47
sko100d	149576	152298	1.81
sko100e	149150	151822	1.79
sko100f	149036	152356	2.22
tai64	1855928	1857650	0.09
tai80	13557864	13856100	2.19
tai100a	21125314	21559700	2.05
tai150b	498896643	510913000	2.40
tai256c	44759294	44898200	0.31
wil50	48816	49002	0.38
wil100	273038	276290	1.19

Table 6: Comparison of TSQAP with other best known fitness from QAPLIB.

overhead to the implementation. Regarding the heuristics from Ehrgott et al., our implementation runs a bit slower, again it's worth mentioning that these programs are Windows application, while our program was run over a Linux platform.

instance	#edges	JLTS Implement.		TSKtree Implement.	
		Fitness	Time t/s	Fitness	Time t/s
g25-4-01.dat	50	219	46	219	1.75
g25-4-02.dat	48	607	53	607	1.50
g25-4-03.dat	50	464	45	464	1.29
g25-4-04.dat	48	620	50	620	1.44
g25-4-05.dat	50	573	50	573	1.60
g50-4-01.dat	98	466	51	460	1.49
g50-4-02.dat	99	421	51	421	1.53
g50-4-03.dat	99	580	51	577	1.41
g50-4-04.dat	100	434	51	434	1.21
g50-4-05.dat	100	398	52	391	1.35
g75-4-01.dat	149	366	52	366	1.61

instance	#edges	JLTS Implement.		TSKtree Implement.	
		Fitness	Time t/s	Fitness	Time t/s
g75-4-02.dat	150	307	52	295	1.63
g75-4-03.dat	149	421	53	421	1.63
g75-4-04.dat	150	432	53	432	1.65
g75-4-05.dat	150	284	54	284	1.49
g100-4-01.dat	200	363	55	363	1.72
g100-4-02.dat	200	338	55	340	1.92
g100-4-03.dat	200	412	55	412	1.89
g100-4-04.dat	199	442	57	442	1.79
g100-4-05.dat	199	388	56	397	1.84
g200-4-01.dat	400	348	61	308	2.61
g200-4-02.dat	400	299	62	299	2.56
g200-4-03.dat	400	300	62	300	2.58
g200-4-04.dat	399	310	62	312	2.43
g200-4-05.dat	399	380	65	357	2.96
g400-4-01.dat	800	253	82	253	2.35
g400-4-02.dat	799	338	74	349	5.70
g400-4-03.dat	799	302	76	306	2.43
g400-4-04.dat	800	315	76	314	2.30
g400-4-05.dat	800	320	78	328	2.37
g1000-4-01.dat	2000	270	110	267	2.85
g1000-4-02.dat	1999	321	120	292	2.62
g1000-4-03.dat	2000	295	124	308	2.79
g1000-4-04.dat	2000	306	120	312	7.19
g1000-4-05.dat	2000	284	131	280	2.59

Table 7: Comparison of our TSKTree with JLTS implementation.

instance	k	DPT Heuristic Fitness	DDT Heuristic Fitness	TSKtree Fitness
gr30_10	3	12	12	12
gr30_10	4	18	18	18
gr30_10	5	27	25	25
gr30_10	6	32	31	31
gr30_10	7	36	36	36
gr30_10	8	42	42	42
gr30_10	9	49	49	49
gr30_10	10	58	54	54
gr30_10	11	68	68	60
gr30_10	12	72	67	67
gr30_10	13	76	76	76
gr30_10	14	83	83	83
gr30_10	15	92	92	92
gr30_10	16	102	102	102
gr30_10	17	109	109	109
gr30_10	18	118	118	118
gr30_10	19	128	128	128
gr30_10	20	139	139	139
gr30_10	21	149	149	149
gr30_10	22	160	160	160
gr30_10	23	171	171	171
gr30_10	24	184	184	184
gr30_10	25	200	200	200
gr30_10	26	218	218	218
gr30_10	27	234	234	234
gr30_10	28	252	252	252

Table 8: Comparison of our TSKTree with Ehrgott et al. heuristics.

instance	k	DPT Heuristic		DDT Heuristic		TSKtree	
		Fitness	t/s	Fitness	t/s	Fitness	t/s
gr30_20	3	11	0.05	11	0.05	11	0.068
gr30_20	4	18	0.06	18	0.05	18	0.077
gr30_20	5	22	0.06	22	0.05	22	0.088
gr30_20	6	30	0.06	30	0.05	30	0.10
gr30_20	7	35	0.05	35	0.05	35	0.11
gr30_20	8	41	0.05	31	0.06	40	0.12
gr30_20	9	46	0.05	46	0.06	46	0.14
gr30_20	10	53	0.05	53	0.06	53	0.16
gr30_20	11	59	0.05	59	0.06	59	0.17
gr30_20	12	66	0.11	66	0.06	66	0.18
gr30_20	13	73	0.11	73	0.10	73	0.20
gr30_20	14	80	0.11	80	0.05	80	0.23
gr30_20	15	89	0.11	88	0.11	88	0.25
gr30_20	16	98	0.17	95	0.11	95	0.27
gr30_20	17	104	0.11	104	0.11	104	0.32
gr30_20	18	113	0.17	113	0.17	13	0.35
gr30_20	19	122	0.16	122	0.17	122	0.39
gr30_20	20	139	0.16	139	0.17	139	0.47
gr30_20	21	155	0.22	155	0.17	155	0.50
gr30_20	22	172	0.22	172	0.16	172	0.60
gr30_20	23	189	0.27	189	0.22	189	0.65
gr30_20	24	208	0.28	208	0.22	208	0.72
gr30_20	25	230	0.28	230	0.22	230	0.97
gr30_20	26	249	0.28	249	0.27	249	0.79
gr30_20	27	275	0.33	275	0.28	275	0.96
gr30_20	28	316	0.33	316	0.28	316	1.36

Table 9: Comparison of our TSKTree with Ehrgott et al. heuristics.