A C++ Implementation of Tabu Search for k-Cardinality Tree Problem Based on Generic Programming and Component Reuse*

M. Josep Blesa and Fatos Xhafa

Departament de LSI
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona, 1-3
08034-Barcelona, Spain.
E-mail: {mjblesa,fatos}@lsi.upc.es

Abstract

We present a C++ implementation of Tabu Search method for the Minimum k-Cardinality Tree problem. Tabu Search is a well known heuristic for sub-optimally solving optimization problems. This method applies also to Minimum k-Cardinality Tree –an important problem to both theory and industrial applications. Our implementation is based on generic programming and re-usability of several components for Tabu Search method. We have compared our program with other *ad hoc* implementations for the problem and have observed a good performance of our implementation. Moreover, our implementation represents time savings, flexibility and robustness mainly due to the component reuse.

Keywords: *k-Cardinality Tree Problem, Tabu Search, Generic Programming, Reuse, Components, Optimization.*

1 Introduction

Generic programming and reuse of components have already proved very important to the software process in general. Interestingly, generic programming and re-usability are also taking place in the design and implementations of programs for solving several problems arising from combinatorial optimization. Such an increasing interest in generic implementations even from the area of combinatorial optimization, where the researchers tends to use *ad hoc* implementations for the problems of interest, can be understood from the advantages of the generic programming and reuse of components such as time savings, flexibility,

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

robustness etc. In combinatorial optimization we are often found with *methods* for sub-optimally solving a large number of optimization problems and such methods apply almost in the same way to all problems. Consider, for example, the Tabu Search method, a well known meta-heuristic [Glo86] that has proved successful in sub-optimally solving a whole bunch of optimization problems. The ingredients of such method are the same for any problem to which one would like to apply the 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. This approach have been considered in our recent work [BX00] where we give a *skeleton* for the Tabu Search method implemented in C++. In brief, there are two basic parts of the skeleton: first, a generic part consisting of several classes to be implemented by the user for the problem at hand, and, second, an implemented part consisting of several reusable components for Tabu Search.

In this paper we show how to obtain an implementation of Tabu Search method for a concrete problem, namely the k-Cardinality Tree, more precisely we show a C++ implementation for the problem based on the *generic skeleton* for Tabu Search. The k-Cardinality Tree problem consists in finding, in a given undirected weighted graph, a subtree of k edges having minimal weight among all of k-trees of the graph. It is a well known combinatorial optimization problem first described by Hamacher, Jörnsten and Maffioli in [HJM91] and it is important to both theory and industrial applications, such as oil field leasing [HJ93]. Other applications of this problem can be found in diverse areas such as facility layout and graph partitioning [FH92, FHW98], open pit mining [FHW97], quorum-cast routing [TCK94], telecommunications [GH97] and matrix decomposition [BFM98, BFM97].

Due to its interest, the problem has been extensively studied and a general complete mathematical formulation of the problem can be found in [JL97] and [EFHM97]. From a complexity point of view, the k-Cardinality Tree was shown to be strongly NP-hard [HJM91] and hence it is unlikely to be solved efficiently. Given this observation, researchers have focused on heuristic algorithms to sub-optimally solve the problem, as a good way to cope in practice with the problem. Among several heuristics applied to the problem, there is also the Tabu Search method. The first to apply Tabu Search method to k-Cardinality Tree were Jörnsten and Løkketangen in [JL97].

We give a new implementation of the Tabu Search method for the k-Cardinality Tree. One can naturally ask why this new implementation? Our main motivations were, first, to obtain a new implementation with as *little effort* as possible, and, second, we wanted our implementation be *flexible*, that means we could easily change several parts of the implementation without affecting the implementation as a whole. We remark that the requirement on the flexibility is very important in the context of Tabu Search method since several ingredients of the method can be implemented in different ways. In order to match this requirements, we address an implementation for the problem based on a *generic skeleton* for the Tabu Search method. 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 our implementation even though obtained through a generic skeleton, it achieves comparable results to those of the specific implementation [JL97] and in some instances better ones.

The paper is organized as follows. In section 2 we give the main points of the Tabu Search skeleton and identify its reusable components. Then, in Section 3 we explain how the Minimum k-Cardinality Tree problem can be easily implemented with our TS skeleton. We make a comparison of our results with the ones in [EFHM97, JL97] in Section 4. We also include more extended experimental results in Appendix A.

2 Tabu Search Skeleton and Its Reusable Components

Tabu Search method starts from an initial solution and *moves* 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.

The TS Skeleton. The design of the skeleton for the Tabu Search was done after a careful review of different known implementations in the literature for the method. We have identified the following *basic ingredients* of the Tabu Search method:

- The **problem** to be solved.
- The form of the **solution** to be reached.
- The **neighborhood** of a given solution.
- The form of the **move** that defines the transition between the current solution and the next one in the neighborhood.
- The **Tabu List**.
- How to **intensify** and **diversify** the search.
- The **main procedure** implementing the Tabu Search algorithm.

Next, these elements are introduced into C++ classes according to a logical definition in the context of the Tabu Search method. Some of these elements have directly become C++ classes (e.g. problem, solution and move). Others have been introduced into classes (e.g. diversification) that are going to be implemented by the user.

The basic idea behind the skeleton is to allow the user to instantiate any combinatorial problem of interest by only defining the most important problem-dependent features. Elements related to the inner algorithmic functionality of the method are hidden to the user. The classes forming the skeleton are divided into a public and a private part. The private part contains elements related to the inner functionality of the TS method, and the public part contains elements related to the problem-dependent part. The public part must be completed by the user, so after filling the problem-dependent features, the problem can be solved using the Tabu Search method. The generality of our skeleton runs in two directions: first, the Tabu Search skeleton permits the user to instantiate any problem of interest by simply *filling in* the public part and second, the skeleton itself is generic enough to solve any problem if its problem-dependent elements can be described.

Reusable Components. Although the general idea of the Tabu Search algorithm is well known (i.e. [HTW97]) some details remain un-standardized. These details are mainly the intensification and diversification of the search. In the literature both are freely introduced *ad hoc* to the schema depending on the problem to solve and the author; that leads to subtle different TS algorithms. Because we wanted a unique generic and powerful enough form of the Tabu Search algorithm, we introduced these functionalities to the basic Tabu Search schema (see figure 1). So a complete standard form of the algorithm was defined and used henceforth.

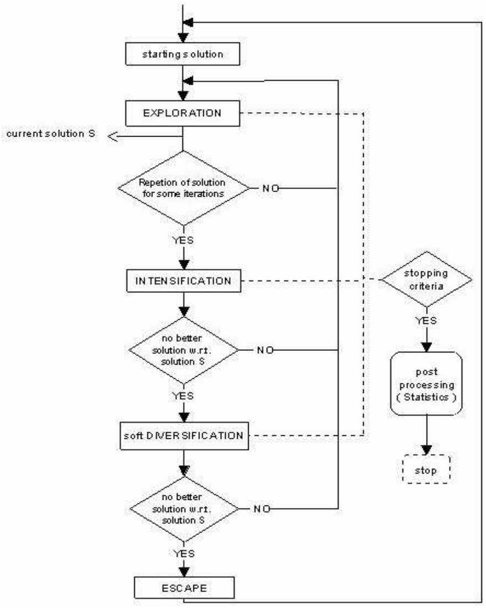

Figure 1: Main procedure of Tabu Search method.

So we considered that the first step to construct a generic form of the Tabu Search method was the standardization and generalization of the main procedure. The second step shares the same objectives but refers to the problem. All the implementations of TS in the literature for different problems are *ad hoc* and quite dependable on the problem at hand. This *ad hoc* strategy 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. That is completely opposite to the generic strategy we wanted to establish.

As mentioned before, the TS skeleton is separated into a public and a private part. The private part implements the generic Tabu Search algorithm and it is completely reusable for all the instantiations. Users should not modify this part in order to not change the TS algorithm functioning. In a certain sense, the main procedure is the most important reusable component of our approach. In order to instantiate a concrete problem the public part must be completed by the user with the problem-dependent features. This is done, for example, by defining (and implementing) what the problem is, how is a solution, what is a move, etc. This definitions can be reused from other instantiation, i.e. both 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.

Different implementations for the same problem can also be generated by varying some definitions and structures from an already implemented version. For example, by changing only the Tabu List structure the management of the tabu moves can be completely different, even its efficiency. Other possible interesting variation are obtained by changing the intensification and/or the diversification strategy to be applied during the search. Only by changing these elements different regions in the space of solutions could be explored, and the only thing the user has to do is to redefine a couple of methods in a class.

So 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.

3 The Implementation for the k-Cardinality Tree

The main ingredients of Tabu Search for k-Cardinality Tree are as follows.

Problem 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 The solution is a k-tree obtained by fixing a stopping criteria, e.g. the number of iterations to be done by the algorithm.

Neighborhood 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 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 We maintain two lists, one for the dropped edges and another one for the added ones.

Intensification We intensify by rewarding those edges of minimum weight that have occurred in the local optima.

Diversification 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.

The instantiation process. 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);
};
```

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;
}
```

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++.

4 Comparison Results and Conclusions

In order to evaluate the performance of our instantiation we have run the instantiated program TSKTree (TS for k-Cardinality Tree) 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).

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 is given by Jörsten and Løkketagen [JL97] 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. [EFHM97] including an extensive list of tests as well executable code consisting of several heuristics for solving the problem.

Jörsten and Løkketagen benchmark. Table 1 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 [JL97] 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 [JL97]) 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 have 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.

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 1: Comparison of our TSKTree with JLTS implementation.

Ehrgott et al. benchmark. Ehrgott et al. [EFHM97] 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 [EFHM97]) 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 2, the results on other tests of the benchmark can be found in Appendix A.

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 2: Comparison of our TSKTree with other heuristics.

We have observed that 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). We have evidenced that our program runs faster than the Jörsten and Løkketagen implementation and a bit slower than the Ehrgott et al. heuristics.

Acknowledgment

We thank the Barcelona Team of Mallba Project for support while conducting this research. We are grateful to A. Løkketangen, M. Ehrgott and H. Hamacher for providing us with their benchmarks and executable codes for k-Cardinality Tree.

References

[BFM97] R. Borndörfer, C. Ferreira, and A. Martin. Matrix Decomposition by Branch-and-Cut. Technical Report SC-97-14, Konrad-Zuse-Zentrum für Informatiosntechnik, Berlin., 1997.

[BFM98] R. Borndörfer, C. Ferreira, and A. Martin. Decomposing Matrices Into Blocks. *SIAM Journal on Optimization*, 9(1):236–269, 1998.

[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.

[EFHM97] M. Ehrgott, J. Freitag, H. Hamacher, and F. Maffioli. Heuristics for the k-Cardinality Tree and Subgraph Problems. *Asia-Pacific J. of Op. Res.*, 14:87–114, 1997.

[FH92] L. Foulds and H.W. Hamacher. A New Integer Programming Approach to (Restricted) Facilities Layout Problems Allowing Flexible Facility Shapes. Technical Report 1992-3, Dept. of Management Science, University of Waikato, 1992.

[FHW97] L. Foulds, A. Hamacher, H.W.Philpott, and N. Wormald. On the Optimal Extraction of Ore from an Open-Cast Mine. Technical report, University of Auckland, New Zeland, 1997.

[FHW98] L. Foulds, H. Hamacher, and J. Wilson. Integer Programming Approaches to Facilities Layout Models with Forbidden Areas. *Annals of Operations Research*, 81:405–417, 1998.

[GH97] N. Garg and D. Hochbaum. An $O(\log k)$ Approximation Algorithm for the k-Minimum Spanning Tree Problem in the Plane. *Algorithmica*, 18:111–121, 1997.

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

[HJ93] H.W. Hamacher and K. Jörnsten. Optimal Relinquishment According to the Norwegian Petrol Law: A Combinatorial Optimization Approach. Technical Report 7/93, Norwegian School of Economics and Business Administration, 1993. submitted to Applied Mathematical Modelling.

[HJM91] H.W. Hamacher, K. Jörnsten, and F. Maffioli. Weighted K-Cardinality Trees. Technical Report 91.023, Dept. di Elettronica, Politecnico di Milano, 1991.

[HTW97] A. Hertz, E. Taillard, and D. de Werra. *Local Search in Combinatorial Optimization.*, chapter 5, pages 121–136. Emile Aarts and Jan Karel Lenstra Ed., 1997. John Wiley & Sons Ltd.

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

[TCK94] L. Trevisan, S. Cheung, and A. Kumar. Efficient Quorumcast Routing Algorithms. In *Proceedings of INFOCOM 94*, pages 840–847. IEEE Society Press, Los Alamitos, 1994.

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. In Table 3 we give results concerning the benchmark of Jörsten and Løkketagen and in Table 4 and Table 5 the ones for (part of) the benchmark of Ehrgott et al. Table 3 and Table 5 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 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 applications, 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
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

instance	#edges	JLTS Implement.		TSKtree Implement.	
		Fitness	Time t/s	Fitness	Time t/s
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 3: 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 4: 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 5: Comparison of our TSKTree with Ehrgott et al. heuristics.