Local search algorithm for unicost set covering problem

Nysret Musliu

Vienna University of Technology, Karlsplatz 13, 1040 Vienna, Austria

Abstract. The unicost set covering problem is a NP-hard and it has many applications. In this paper we propose a new algorithm based on local search for solving the unicost set covering problem. A fitness function is proposed for this problem and different neighborhood relations are considered for the exploration of the neighborhood of the current solution. A new approach is introduced for effective exploration of the neighborhood during the improvement phase. This approach is based on the upper bound of the best cover, which is found during the search, and using only determined moves. Additionally, in order to avoid cycles during the search, a search history is used. The proposed algorithm is experimentally evaluated for 85 well-known random and combinatorial problems in the literature, and it gives very satisfactory results in a reasonable amount of time. The proposed algorithm improves the best existing solutions for 8 problems in the literature. For a class of combinatorial problems, the best existing results are improved significantly.

1 Introduction

For the set covering problem ([7]) we have given the finite set X and the family F of subsets of X. Each element of X belongs to at least one subset of F. The problem is to find the minimum size subset $C \subseteq F$ whose members cover all elements of X. The variant of the set covering problem in which all sets in F have the weight 1 is called the unicost set covering problem.

Many problems can be formulated as a set covering problem. Applications of a set covering problem include scheduling, testing, construction of optimal logical circuits, inspection of computer viruses, etc. The unicost set covering problem considered in this paper appeared in the problem of generation of generalized hypertree decompositions from tree decompositions. The importance of tree decompositions and hypertree decompositions lies in the fact that many constraint satisfaction and other NP-complete problems can be polynomially solved if their associated hypergraph has a low width for the chosen decomposition. Tree and hypertree decompositions are out of the main focus of this paper, and therefore the reader is refereed to [5] and [10] for definitions and further details on the importance of these concepts for solving constraint satisfaction problems.

Different methods have been proposed in the literature for the set covering problem. Examples of using exact methods are [1] and [2]. For solving larger instances of this problem, approximation algorithms and heuristic algorithms

have been proposed. One of the best approximation algorithms is the greedy algorithm [6]. The greedy algorithm finds a solution for the set covering problem by iteratively picking a set that covers as many remaining uncovered variables as possible. Grossman and Wohl [11] compared nine different algorithms for the set covering problem including different variants of the greedy algorithm, randomized algorithms, and neural networks. They reported that the best results were obtained by using the randomized greedy algorithm. This algorithm is identical to the greedy algorithm, except that in the phase of selection ties are broken randomly for sets which contain the same number of uncovered variables. The results obtained by the randomized greedy algorithm were obtained by iterating this algorithm 100 times.

Marchiori and Steenbeek [13] enhanced the greedy algorithm by introducing a new rule for breaking ties in the steps of adding and removing sets. Further, the simple optimization step was applied in the constructed solution by the enhanced greedy algorithm. An iterated algorithm is proposed that starts each run with partial cover obtained from the solution constructed by the enhanced greedy algorithm in the previous iteration. The authors reported very good results with this method and they improved the best results for many problems in the literature. Nonobe and Ibaraki [14] have proposed a tabu search algorithm for constraint satisfaction problems. This general tabu search problem solver has been applied for the set covering problem and the algorithm has been evaluated for the class of problems defined from Steiner triple systems. The tabu search approach has also been used in [12], here the authors compared their results for 70 problems to CPLEX, and report better results than those reported in [11]. However, the quality of results reported in [12] are worse than the results reported in [13]. Other metaheuristic approaches, such as genetic algorithms ([4]) have been proposed for solving this problem.

In this paper we propose a new method based on local search for solving the unicost set covering problem. We investigate the use of different neighborhood relations and propose a simple fitness function for this problem. The proposed algorithm includes a new method for the generation of the neighborhood during each iteration based on the knowledge of the number of sets of the current best known solution. Additionally, the algorithm uses a search history and a basic tabu search mechanism to avoid cycles during the search.

This paper is organized as follows: in the next section the local search algorithm for solving the unicost set covering problem is described. In section 2, its computational results on 85 problems are presented, before the final section which contains our conclusions and remarks.

2 Local search algorithm

In this section a local search based algorithm for unicost set covering problem is described. The algorithm is based on an iterative local improvement of the initial solution. In each iteration the neighborhood of the current solution is generated, and one solution from this neighborhood is selected for the next

iteration. A basic variant of a local search is used for the selection criteria, in which the best solution of the neighborhood is selected for the next iteration. However, also the so called tabu mechanism to avoid the cycles during the search is introduced. If the solutions are obtained by moves, which are performed in a determined number of previous iterations, then they are considered as tabu solutions and will be accepted only if they fulfill particular criteria. For the generation and restriction of the neighborhood of the current solution, a unique method is applied. This method is based on the upper bound for the number of sets of the best solution found during the search. The basic elements of this local search procedure are described below in detail.

2.1 Neighborhood structure

As described in Section 1 the solution for the set covering problem is a subset $C \subseteq F$ whose members cover all elements of X. Note that the solution, which in this case is the subset C, can be illegal in the beginning and also in the improvement phase, i.e. a solution which does not cover all elements of X. To generate the neighborhood of subset C two basic moves are applied:

$ADD\_SET(S)$ - a set S (which is not in C) from the family of sets F is added in C: $C \leftarrow C \cup S$

$REMOVE\_SET(S)$ - a set S is removed from C: $C \leftarrow C - S$

Another move, which has been used in the literature, swaps one set $S1$ in C with another set $S2$ not in C: $SWAP\_SETS(S1, S2)$

The whole neighborhood that is generated by only applying ADD_SET and $REMOVE\_SET$ moves contains $|F|$ solutions. If the $SWAP\_SETS$ move is applied the generated neighborhood is much larger. In this case the whole neighborhood contains $|C| * (|F| - |C|)$ solutions.

The $SWAP\_SETS$ move can be performed by the sequence of ADD_SET and $REMOVE\_SET$ moves:

$$SWAP\_SETS(S1, S2) = REMOVE\_SET(S1) + ADD\_SET(S2)$$

To avoid a very large number of solutions in the neighborhood we implemented a method which only applies ADD_SET and $REMOVE\_SET$ moves. We also experimented with all three moves mentioned above. The experiments showed that the moves ADD_SET and $REMOVE\_SET$ are sufficient to obtain very satisfactory solutions in a reasonable amount of time.

2.2 Restricting neighborhood by introducing an upper bound parameter

We define the upper bound parameter as the number of sets of actual best legal solution. If only the ADD_SET and $REMOVE\_SET$ moves are applied, this upper bound can be used to restrict the neighborhood during the search. With this restriction the move ADD_SET can be applied in the current solution only if the number of sets in the current solution is less than $upperbound - 1$.

The upper bound is calculated from the first legal solution, which covers all elements of X. If the legal solution is improved during the search, the upper bound parameter is updated automatically. The motivation for restricting the neighborhood during the search is given below. By using this restriction the search is intensified in the neighborhood (considering number of sets) of the current best solution by not allowing the exploration of the solutions which have more sets than the current best legal solution. However, this method does not restrict the exploration of solutions with worse fitness. Moreover, by using fitness function defined in section 2.3, with this restriction is enforced almost all the times during the improvement phase (the phase in which one cover is already known) that in one iteration the $REMOVE\_SET$ is performed, whereas in the next iteration ADD_SET move is performed. The move $REMOVE\_SET$ can be applied during each iteration, if the variables can be covered with less sets. Through alternate applying of ADD_SET and $REMOVE\_SET$ we mimic the $SWAP\_SETS$ move with much smaller neighborhood during each iteration.

Additionally, to make the search more effective, during the neighborhood exploration the following rule is applied: if the set is removed from C, in the next iteration for adding in C are considered only the sets which share elements with the set removed in the previous iteration. This requires storing of additional information for the problem, but makes the search more effective.

2.3 Fitness function

A simple fitness function is used to calculate the quality of solution during the search. A penalty of value 1 for each uncovered element of set X and each set in C is given. The fitness of solution is equal with the sum of uncovered variables and the number of sets in C:

$$Fitness = NrOfUncoveredElements + |C|$$

The aim is to minimize this fitness function during the search, such that all elements of set X are covered.

To calculate the fitness efficiently, the fitness of neighborhood solutions is calculated only from the fitness of the previous solution and the change caused from the applied move.

2.4 Initial solution

We experimented with the empty initial solution and the initial solution which is generated with the greedy algorithm [6]. The greedy algorithm finds a solution for the set covering problem by iteratively picking a set that covers as many remaining uncovered variables as possible. The results presented in this paper are obtained with the initial solution which is obtained by the greedy algorithm.

2.5 Using of the search history

In order to avoid the cycles during the search, the search history is used. The information for the search are stored in a short term memory (tabu list). In this memory are stored the information for the sets which are removed or added in the past number of iterations $nIter$. For example, if the solution accepted for the next iteration is obtained by adding the set $S1$ in C, in the tabu list will be added the set $S1$. The moves $ADD\_SET(S1)$ and $REMOVE\_SET(S1)$ are made tabu (can not be applied) for several iterations depending on the length of the tabu list. We experimented with the different lengths for the tabu list. In the first variant the length of the tabu list is same for all examples independently from the size of the problem. In the second variant we experimented with the tabu length which depends from the number of sets in the solution generated by the greedy algorithm. Experimental results showed that the second variant gives better results for the problems we consider in this paper. The information for the length of the tabu list used in experiments are given in Section 3.

2.6 Selection criteria

The solution is accepted for the next iteration based on the following criteria: the best solution (ties are broken randomly) from the neighborhood becomes the current solution in the next iteration, if it is not tabu solution. The solution is considered tabu, if it is obtained by applying of one of moves on the set which is in the tabu list. If the best solution from neighborhood is tabu, the aspiration criterion is applied. For the aspiration criteria, we use a standard version [9] according to which the tabu status of a move is ignored if the move has a fitness better than the current best solution.

2.7 The algorithm

The pseudo code of the overall local search algorithm which was shown the best in the experimental evaluation is given below. This algorithm exploits the simple fitness function defined in the section 2.3, applies only ADD_SET and $REMOVE\_SET$ moves, and makes use of the short term memory. Additionally, the upper bound parameter is used to reduce the neighborhood.

Local search algorithm
1. Generate the initial solution
2. Initialize the tabu list and the UPPERBOUND parameter
3. Generate the neighborhood of the current solution as described in section 2.2 (apply only ADD_SET and $REMOVE\_SET$ moves)
4. Evaluate the neighborhood solutions
5. Select the solution for the next iteration (see section 2.6)
5. Update the tabu list and UPPERBOUND parameter
7. Go to step 3 if the stopping criteria is not fulfilled, otherwise go in step 8
8. Return the best legal solution

3 Computational results

The algorithms proposed in this paper are evaluated experimentally for 85 problems from the literature. These problems include random problems and the set covering problems which appeared in different combinatorial problems. Characteristics of these instances are presented in [13]. Instances 4-6, A-E and NRE-NRH are random generated instances. Instances 4-6 are from [1], A - E from [2], and NRE - NRH from [3]. All these instances are available from the OR-library (http://mscmga.ms.ic.ac.uk/jeb/orlib/scpinfo.html). These instances represent a weighted set covering problem, in which columns can have different costs. As in [11], [13], [12] the costs of columns are discarded when applying the algorithm to the unicost set covering problem. The results obtained in this paper are compared with the results from [11], [13], [12] and are not comparable with (except for instances $E.1 - E.5$) the results obtained by [1], [2], and [3] because they considered a weighted set covering problem.

The CYC and CLR instances are from [11] and arise from combinatorial problems. These instances are derived from two questions of Erdós and are described in detail in [11]. The STS instances are derived from Steiner triple systems ([8]), and they are known to be difficult set covering instances.

Our algorithms are implemented in C++ and the experiments are performed in a computer Pentum 4, 2,4GHZ, 512 MB RAM. For each problem 10 independent runs are executed.

The algorithm presented in Section 2.7, which has been shown to be very powerful for unicost set covering problem exploits the tabu list. The parameter for the length of the tabu list is found experimentally for each class of instances mentioned before. The tabu length depends from the number of sets in the solution that is obtained by the greedy algorithm (we denote this parameter as NrOfGreedySets). For each class of problems the length of the tabu list is obtained as follows: $TabuLength = TSFactor * NrOfGreedySets + 1$. The experiments with the following tabu length factor ($TSFactor$) were performed: 0, 0.05, 0.1, 0.15 and 0.2. The best tabu length for each class of problems was chosen based on the sum of the average number of sets in 10 runs (for all solutions of that particular class), and based on the average time for which the solutions are found.

In Tables 1, 2, and 3 are presented the results of the algorithm proposed in this paper for 85 problems from the literature. The results obtained with the previous approaches proposed in the literature are also presented in these tables. The ITEG algorithm [13] was tested on a multi-user Silicon Graphics IRIX Release 6.2 IP25, 194 MHZ MIPS R10000 processor, 512 MB RAM. The algorithms from [12] were tested on Dell Precision 530 Workstations with two 1.8 GHz Pentium Xeon processors and 1GB of RAM. First column of each table represents the problems for which the results are given. In the second column for each problem is presented the best solution obtained by the algorithms proposed in [11] (the number represents number of sets the solution contains). In the third and the fourth column are presented results from [12]. The best solution is presented in the third column, whereas the time for which the solution is found

is presented in the fourth column (T). Next three columns represent results from [13] (best solution, averages number of sets for 10 runs, and average time used for each problem). The last three columns give the results obtained by the algorithm proposed in this paper (see section 2.7). From these three columns the first column represents for each problem the best solution obtained in 10 runs. The second column gives the average and standard deviation of number of sets in 10 runs. Last column gives information for the average (and standard deviation) time for which the best solution is found. The time for all algorithms is given in seconds.

In [14] and [15] are given results only for STS class of problems. These results are not included here. To our best knowledge the tables given in this paper represent the best known results for the unicost set covering problems found in the literature. The only exception is the problem STS.135 for which a better solution (with value 103) is reported in [15].

Based on these tables we can conclude that the approach proposed in this paper gives better results compared to [11] for 52 problems. The methods proposed in [11] do not outperform our algorithm for any of problems. Compared to the methods proposed in [12] our approach gives better results for 24 problems. The only problem for which the approach proposed in [12] is better (for 1 set) is the problem *scpnrg*2. In [12] are not given results for CYC, CLR and STS problems. The ITEG algorithm proposed in [13] outperforms slightly our approach (considering average in 10 runs) in 22 examples, whereas our approach outperforms ITEG in 34 problems. Considering the best solution in 10 runs our approach gives better results for 9 problems compared to ITEG, and ITEG does not give better results than our approach for any of problems. The algorithm proposed in this paper improves the best known results in the literature for 8 problems (6.4, A.4, D.3. NRG1, CYC08 - CYC11) and can find the best known results for all problems, except for the problems NRG2 (62 instead of 61 obtained in [12]) and STS.135 (104 instead 103 obtained in [15]). The results for problems $CYC.08 - CYC.11$ are improved significantly by our algorithm. Note that with the individual tabu length for each problem (not the tabu length for the class of problems) our algorithm could also find better solution for the following problems: $NRE4(16)$, $A.2(38)$, and $D.1(24)$.

4 Conclusion

In this paper we presented a local search algorithm for solving the unicost set covering problem. We have used simple neighborhood relations and have introduced a unique approach for neighborhood exploration during the search. The neighborhood exploration is based on the upper bound of the number of sets in the best solution and restriction of particular add moves. This restriction is based on information about the removed set in the previous iteration. Additionally, we have used a search history to avoid possible cycles during the search. The proposed algorithm has been evaluated experimentally in 85 problems which appeared previously in the literature. Experimental results show that our local

Table 1. Comparison of the results for the class of problems 4,5,6,A,B,C,D,E

	[11]	[12]		ITEG ([13])			Our algorithm		
Example	Best	Best	T	Best	Avg	T(Avg)	Best	Avg(Stdev)	T(Avg/Stdev)
scp41.txt	41	38	938	38	38	10	38	38,1 (0,3)	0,5 (0,7)
scp42.txt	38	37	5	37	37	10	37	37 (0,0)	0 (0,0)
scp43.txt	40	38	1	38	38	10	38	38 (0,0)	0 (0,0)
scp44.txt	41	38	272	39	39,1	10	38	38,6 (0,5)	0,7 (1,1)
scp45.txt	40	38	23	38	38	10	38	38 (0,0)	0,4 (1,0)
scp46.txt	40	37	3	37	37,8	10	37	37,2 (0,4)	0,8 (0,9)
scp47.txt	41	38	413	38	38,4	10	38	38,4 (0,5)	1,1 (0,7)
scp48.txt	40	38	6	37	37,7	10	37	37,6 (0,5)	1 (1,3)
scp49.txt	40	38	35	38	38,1	10	38	38 (0,0)	1 (1,2)
scp410.txt	41	38	161	38	38,6	10	38	38,3 (0,5)	1,2 (1,4)
scp51.txt	35	35	5	34	34,9	10	34	34,7 (0,5)	1 (2,2)
scp52.txt	35	35	6	34	34,7	10	34	34,2 (0,4)	3,2 (2,6)
scp53.txt	36	34	39	34	34	10	34	34 (0,0)	0,8 (1,4)
scp54.txt	36	34	1182	34	34	10	34	34 (0,0)	1,6 (2,0)
scp55.txt	36	34	12	34	34,1	10	34	34,1 (0,3)	2,2 (3,0)
scp56.txt	36	34	989	34	34,5	10	34	34,1 (0,3)	3,1 (3,0)
scp57.txt	35	34	75	34	34	10	34	34 (0,0)	0,6 (1,1)
scp58.txt	37	34	74	34	34,9	10	34	34,4 (0,5)	2,2 (3,6)
scp59.txt	36	35	6	35	35	10	35	35,6 (1,0)	0,6 (1,0)
scp510.txt	36	34	1873	34	34,6	10	34	34,5 (0,5)	3,4 (3,6)
scp61.txt	21	21	5	21	21	60	21	21 (0,0)	0 (0,0)
scp62.txt	21	21	6	20	20,3	60	20	20 (0,0)	0,7 (0,8)
scp63.txt	21	21	10	21	21	60	21	21 (0,0)	0 (0,0)
scp64.txt	22	21	4	21	21	60	20	20,9 (0,3)	0,6 (1,9)
scp65.txt	22	21	25	21	21	60	21	21 (0,0)	0 (0,0)
scpa1.txt	40	39	337	39	39,1	30	39	39 (0,0)	3,2 (2,9)
scpa2.txt	41	39	79	39	39,1	30	39	39 (0,0)	4,7 (3,1)
scpa3.txt	40	39	179	39	39	30	39	39,1 (0,3)	1,8 (1,8)
scpa4.txt	40	38	1715	38	38	30	37	37,8 (0,4)	5,7 (6,6)
scpa5.txt	40	38	771	38	38,7	30	38	38,4 (0,5)	6,1 (4,5)
scpb1.txt	23	22	719	22	22	60	22	22 (0,0)	8,3 (8,8)
scpb2.txt	22	22	17	22	22	60	22	22 (0,0)	2 (4,2)
scpb3.txt	22	22	698	22	22	60	22	22 (0,0)	1,1 (3,5)
scpb4.txt	23	22	1910	22	22	60	22	22,1 (0,3)	11,6 (9,5)
scpb5.txt	23	22	46	22	22,2	60	22	22 (0,0)	12,1 (8,9)
scpc1.txt	45	43	1524	43	43,5	40	43	43,5 (0,5)	5,9 (5,7)
scpc2.txt	45	44	197	43	43,5	40	43	43,4 (0,5)	9,5 (8,5)
scpc3.txt	45	43	1029	43	43,6	40	43	43,4 (0,5)	10,2 (10,4)
scpc4.txt	46	43	1325	43	43,1	40	43	43,3 (0,5)	11,6 (9,2)
scpc5.txt	45	44	149	43	43,5	40	43	43,9 (0,3)	2,1 (2,0)
scpd1.txt	26	25	395	25	25	110	25	25,1 (0,3)	6,4 (10,7)
scpd2.txt	25	25	1890	25	25	110	25	25 (0,0)	2,2 (1,4)
scpd3.txt	25	25	91	25	25	110	24	24,9 (0,3)	21,6 (27,0)
scpd4.txt	26	25	226	25	25	110	25	25 (0,0)	17,7 (16,4)
scpd5.txt	26	25	200	25	25	110	25	25 (0,0)	24,1 (16,4)
scpe1.txt	5	5	-	5	5	10	5	5 (0,0)	0 (0,0)
scpe2.txt	5	5	-	5	5	10	5	5 (0,0)	0 (0,0)
scpe3.txt	5	5	-	5	5	10	5	5 (0,0)	0 (0,0)
scpe4.txt	5	5	-	5	5	10	5	5 (0,0)	0 (0,0)
scpe5.txt	5	5	-	5	5	10	5	5 (0,0)	0 (0,0)

Table 2. Comparison of the results for the class of problems NRE, NRF, NRG, NRH

	[11]	[12]		ITEG ([13])			Our algorithm		
Example	Best	Best	T	Best	Avg	T(Avg)	Best	Avg(Stdev)	T(Avg/Stdev)
scpnre1.txt	17	18	38	17	17	34	17	17,3 (0,5)	2,2 (3,0)
scpnre2.txt	17	18	27	17	17	34	17	17,1 (0,3)	1,5 (2,3)
scpnre3.txt	17	18	32	17	17	34	17	17,1 (0,3)	16,5 (38,8)
scpnre4.txt	17	17	54	17	17	34	17	17,2 (0,4)	5 (7,8)
scpnre5.txt	17	17	586	17	17,2	34	17	17 (0,0)	4,5 (4,6)
scpnrf1.txt	10	11	29	10	10,3	66	10	10,7 (0,5)	17,3 (28,9)
scpnrf2.txt	11	11	39	10	10,4	66	10	10,5 (0,5)	43,9 (63,3)
scpnrf3.txt	11	11	30	10	10,6	66	10	10,6 (0,5)	48,7 (112,6)
scpnrf4.txt	11	11	22	10	10,5	66	10	10,7 (0,5)	17,9 (31,5)
scpnrf5.txt	11	10	482	10	10,7	66	10	10,6 (0,5)	29,4 (73,5)
scpnrg1.txt	-	63	1089	62	62,4	26	61	62,4 (0,8)	27,3 (24,1)
scpnrg2.txt	-	61	3401	62	62,5	26	62	62,3 (0,5)	29,8 (34,4)
scpnrg3.txt	-	62	901	62	62,8	26	62	62,9 (0,6)	20,8 (24,5)
scpnrg4.txt	-	63	1045	62	63,7	26	62	63,1 (0,7)	41,8 (42,7)
scpnrg5.txt	-	63	406	62	62,7	26	62	62,8 (0,4)	40,2 (35,7)
scpnrh1.txt	-	35	2008	34	34,8	61	34	34,9 (0,6)	8,7 (16,3)
scpnrh2.txt	-	36	297	34	34,7	61	34	34,9 (0,3)	7,8 (21,2)
scpnrh3.txt	-	36	968	34	34,8	61	34	34,9 (0,3)	19,1 (32,2)
scpnrh4.txt	-	35	940	34	35	61	34	34,9 (0,6)	26,1 (67,7)
scpnrh5.txt	-	36	454	34	34,6	61	34	34,8 (0,4)	50,3 (150,0)

Table 3. Comparison of the results for the problems CYC, CLR and STS

	[11]	[12]		ITEG ([13])			Our algorithm		
Example	Best	Best	T	Best	Avg	T(Avg)	Best	Avg(Stdev)	T(Avg/Stdev)
scpcyc06.txt	60	-	-	60	61,4	0	60	60 (0,0)	0 (0,0)
scpcyc07.txt	144	-	-	144	148	1	144	144 (0,0)	0 (0,0)
scpcyc08.txt	352	-	-	348	351,8	5	342	343,8 (0,6)	11,1 (10,4)
scpcyc09.txt	816	-	-	825	827,6	19	774	791,9 (6,4)	110,4 (122,6)
scpcyc10.txt	1916	-	-	1858	1860,7	110	1820	1823,9 (2,9)	488,9 (189,7)
scpcyc11.txt	4268	-	-	4202	4218,3	500	4088	4144,9 (24,5)	1497,8 (67,2)
scpclr10.txt	28	-	-	25	25	10	25	25 (0,0)	0 (0,0)
scpclr11.txt	27	-	-	23	23	20	23	23 (0,0)	0 (0,0)
scpclr12.txt	27	-	-	23	23	50	23	23 (0,0)	3,7 (3,8)
scpclr13.txt	29	-	-	23	25,3	110	23	23,4 (0,5)	79 (64,9)
STS27.txt	-	-	-	18	18	1	18	18 (0,0)	0 (0,0)
STS45.txt	-	-	-	30	30,7	2	30	30 (0,0)	0,1 (0,3)
STS81.txt	-	-	-	61	61	11	61	62,4 (1,0)	0 (0,0)
STS135.txt	-	-	-	104	104	15	104	105,7 (1,2)	1,1 (3,5)
STS243.txt	-	-	-	198	202,2	100	198	199 (2,1)	29,6 (31,1)

search algorithm gives very satisfactory results for problems appearing in the literature. In particular, our algorithm performs as well as the best results cited in the literature, except for two problems. The solutions generated by our algorithm for these two problems are only worse with respect to 1 set. For 8 problems from the literature our algorithm discovers better solutions. In the future work we are planning to analyze the use of the frequency based memory and to apply the adaptive tabu list for the unicost set covering problem.

Acknowledgments: This paper was supported by the Austrian Science Fund (FWF) project: *Nr. P17222-N04, Complementary Approaches to Constraint Satisfaction*

References

1. *E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study.* Mathematical Programming*, 12:37–60, 1980.*
2. *J. E. Beasley. An algorithm for set covering problems.* European Journal of Operational Research*, 31:85–93, 1987.*
3. *J. E. Beasley. A lagrangian heuristic for set covering problems.* Naval Research Logistics*, 37:151–164, 1990.*
4. *J. E. Beasley and P. C. Chu. A genetic algorithm for set covering problem.* European Journal of Operational Research*, 94:392–404, 1996.*
5. *H. L. Bodlaender. Discovering treewidth. In* In P. Vojtás, M. Bieliková, B. Charron-Bost, and O. Sýkora, editors, SOFSEM 2005: Theory and Practice of Computer Science, pages 116. Springer, Lecture Notes in Computer Science, vol. 3381, 2005.
6. *V. Chvátal. A greedy heuristic for the set-covering problem.* Math. of Oper. Res.*, 4/3:233–235, 1979.*
7. *T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, editors.* Introduction to Algorithms, 2nd ed. *The MIT Press, Massachusetts, 2001.*
8. *D.R. Fulkerson, G.L. Nemhauser, and L.E. Trotter. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems.* Mathematical Programming Study*, 2:72–81, 1974.*
9. *Fred Glover and Manuel Laguna.* Tabu search*. Kluwer Academic Publishers, 1997.*
10. *G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods.* Artificial Intelligence*, 124(2):243–282, 2000.*
11. *T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem.* European Journal of Operational Research*, 101:81–92, 1997.*
12. *G. Kinney, J.W. Barnes, and B. Colleti. A group theoretic tabu search algorithm for set covering problems.* Working paper, available from http://www.me.utexas.edu/ barnes/research/*, 2004.*
13. *Elena Marchiori and Adri Steenbeek. An iterated heuristic algorithm for the set covering problem.* Proceedings of WEA'98, Germany*, 1998.*
14. *K. Nonobe and T. Ibaraki. A tabu search approach to the constraint satisfaction problem as a general problem solver.* European Journal of Operational Research*, 106:599–623, 1998.*
15. *M.A. Odijk and H. van Maaren. Imporved solutions for the steiner triple covering problems.* Technical Report, TU Delft University, 11, 1996*, 1996.*